ARI Research Note 91-79

Developing a General Contingency Planner for Adversarial Planning

Paul E. Lehner and James R. McIntyre
PAR Technology Corporation

for

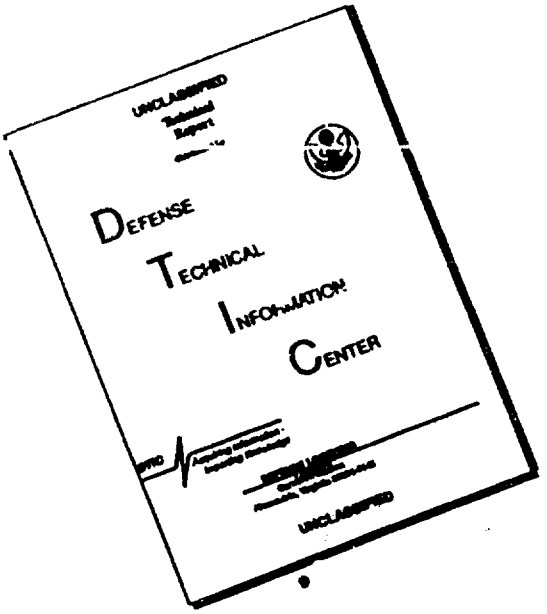
Contracting Officer's Representative
Michael Drillings

Office of Basic Research
Michael Kaplan, Director

June 1991

United States Army
Research Institute for the Behavioral and Social Sciences

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

U.S. ARMY RESEARCH INSTITUTE FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency Under the Jurisdiction of the Deputy Chief of Staff for Personnel

EDGAR M. JOHNSON
Technical Director

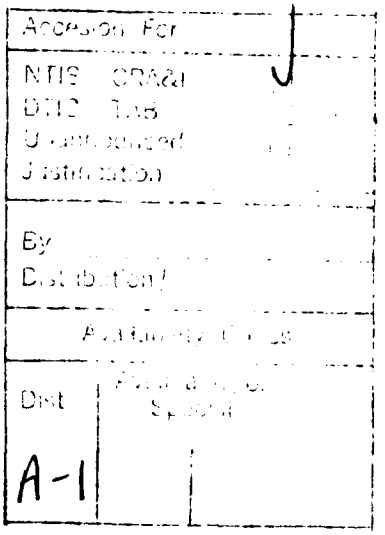
JON W. BLADES
COL, IN
Commanding

Research accomplished under contract
for the Department of the Army

PAR Technology Corporation

Technical review by

Michael Drillings



NOTICES

DISTRIBUTION: This report has been cleared for release to the Defense Technical Information Center (DTIC) to comply with regulatory requirements. It has been given no primary distribution other than to DTIC and will be available only through DTIC or the National Technical Information Service (NTIS).

FINAL DISPOSITION: This report may be destroyed when it is no longer needed. Please do not return it to the U.S. Army Research Institute for the Behavioral and Social Sciences.

NOTE: The views, opinions, and findings in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other authorized documents.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS --
2a. SECURITY CLASSIFICATION AUTHORITY --	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE --	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) PAR 84-126	5. MONITORING ORGANIZATION REPORT NUMBER(S) ARI Research Note 91-79

6a. NAME OF PERFORMING ORGANIZATION PAR Technology Corporation Decision Sciences Section	6b. OFFICE SYMBOL (If applicable) --	7a. NAME OF MONITORING ORGANIZATION U.S. Army Research Institute Office of Basic Research
6c. ADDRESS (City, State, and ZIP Code) 7926 Jones Branch Drive McLean, VA		7b. ADDRESS (City, State, and ZIP Code) 5001 Eisenhower Avenue Alexandria, VA 22333-5600
8a. NAME OF FUNDING/SPONSORING ORGANIZATION U.S. Army Research Institute for the Behavioral and Social Sciences	8b. OFFICE SYMBOL (If applicable) PERI-BR	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA903-83-C-0355
8c. ADDRESS (City, State, and ZIP Code) Office of Basic Research 5001 Eisenhower Avenue Alexandria, VA 22333-5600		10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
61102B	74F	N/A	N/A

11. TITLE (Include Security Classification)
Developing a General Contingency Planner for Adversarial Planning

12. PERSONAL AUTHOR(S)
Lehner, Paul E.; and McIntyre, James R.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM 83/09 TO 84/09	1991, June	87

16. SUPPLEMENTARY NOTATION
Michael Drillings, Contracting Officer's Representative

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Tactical planning; Time pressure; Decision making; Uncertainty

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report summarizes the first year of a 3-year effort to investigate the basic mechanisms for solving planning problems in environments that contain intelligent adversaries. A general purpose planner that can solve adversarial planning problems in a variety of domains is presently under development. This planner, called CP/x (Contingency planner version x), uses a formalism for representing the plans, which are consistent with the goal tree formalisms found in action planning in Artificial Intelligence (AI), while using plan generation/search techniques derived from both AI action planning and "knowledge-based" game playing theory. CP/1.0, the first version of this planner, is described in this report, and approaches to incorporating advanced planning techniques (e.g., metaplanning, hierarchical, and distributed planning) are discussed.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☑ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Michael Drillings	22b. TELEPHONE (Include Area Code) (703) 274-8722
	22c. OFFICE SYMBOL PERI-BR

DD Form 1473, JUN 86 *Previous editions are obsolete.*

TABLE OF CONTENTS

SECTION PAGE

LIST OF FIGURES

1.0 INTRODUCTION

One of the potentially most valuable research areas within Artificial Intelligence (AI), from the perspective of military applications, involves the development of automatic planning systems. AI planners not only have the potential of autonomously solving various low-level planning problems such as in robotics applications, but can form the basis of expert consultation systems for higher level military planning problems, (e.g., planning division level maneuvers).

Most of the work in AI action planning has been done in the context of robot problem solving, where the planners developed in this context can be extended, at least in principal, to other domains. Recent research in this area has focused on the development of techniques to extend action planners to real world complex problem domains. In particular, these efforts have focused on adapting planners to uncertain and unpredictable environments where multiple components of the plan may be executed in parallel by distributed execution modules. Although this work is relevent to the development of practical AI planners, from the perspective of eventual military value, it is inherently limited. In particular, these planners lack a satisfactory capability to explicitly incorporate an adversary's goals and actions into the planning process. Consequently, they cannot effectively plan against an adversary that is simultaneously planning against them.

To illustrate the importance of incorporating an adversary's goals, consider the work in planning under uncertainty. One accepted technique is the use of information goals, where if information

necessary to execute a plan is not available to the planner, then a goal of collecting the required information is added to the plan, (e.g., FIND_LOC(x), if location of x is not known). When an adversary is present, however, that adversary is likely to have a counterplan to prevent the collection of the required information, and may use any of a variety of tactics (e.g., have a goal of DECEIVE_LOC(x)) to achieve this end. Unless the adversary's counter goals and actions are explicitly taken into account, the information goal is not likely to be achieved.

In this report we discuss the problem of extending action planning techniques to problems involving planning against an intelligent adversary. In particular, this report summarizes the work performed in the first year of a research program to develop automated adversarial planning techniques. Section 2.0 provides some background on AI planning and game playing research. Section 3.0 describes in some detail, the capabilities and planning procedures of CP/1.0 (Contingency Planner/Version 1.0), our first version of a general adversarial planning system. In addition, Section 3.0 also discusses each individual action planning technique discussed in Section 2.0, and discusses how they might be incorporated into the CP/x framework and therefore extended to adversarial planning problems. Section 4.0 briefly addresses our plans for the next two years of this program.

2.0 BACKGROUND

In this section, research relevant to the development of adversarial planning techinques is reviewed.

2.1 Single Agent Action Planning

Most of the artificial intelligence (AI) work in planning has been done in the context of research where a single-agent (e.g., a robot) must plan a sequence off action. Since the goal of this work is to extend planning techniques to adversarial planning problems, relevant action planning techniques are introduced below.

2.1.1 Goal-Driven Action Planning

Within AI research, a common planning task involves a robot (real or hypothetical) that exists in some type of blocks world. The robot is given tasks which require it to move the blocks into some particular configuration. Before starting, the robot is required to 'figure out' what actions need to be taken, i.e., to make a plan.

Some early planners (e.g., Fikes, et. al., 1972) would solve problems such as this through a 'backward chaining' procedure. In this approach, the planner is given a set of goals that are sufficient to describe the goal state, a description of the initial state of the world and a set of permissable operators or actions that can modify the world state. The planner then looks at each goal, matches it against the present world state and if it is not already true, looks for an operator that can achieve it. If the operator cannot be applied, because a precondition of its use is not satisfied, then the unsatisfied precondition becomes a new goal to be achieved. This process is recursively repeated until all goals are satisified.

To illustrate this approach, assume that we have a robot that exists in a world consisting of three square blocks and a table. Assume further that the blocks are initially set up as in Figure 2-1. The robot is capable of only one action, PUTON(x,y), which means pick up block x and put it on top of object y. y can either be another block or the table. PUTON can only move one block at a time. Assume further that our robot has been given the task of moving the blocks so that block A is on top of B which is on top of C (see Figure 2-2).

This goal state can be described by the conjunction of goals ON(B,C) and ON(A,B). Assume our robot has been given these goals, and it begins by trying to attain the first one, ON(B,C). Now, in order to obtain this goal, the action PUTON(B,C) must be executed. However, this action cannot be executed until the precondition that 'C is clear' is true. Consequently, CLEAR(C) becomes a new goal to be achieved. PUTON(C,Table) can directly achieve this and is consequently the first action in the plan. Once C is clear, PUTON(B,C) is now executable and therefore becomes the second action. At this point ON(B,C) has been achieved, and ON(A,B) which is executable. Consequently, this becomes the third action in the plan, at which point the plan is complete.

One way to record this planning process is by the use of a hierarchical goal tree where each goal, subgoal and action, that is part of the plan, corresponds to a node in the tree. For example, a goal tree that corresponds to the plan generated in the above example is found in Figure 2-3. There are two features of this goal tree that should be noticed. First the sequence of actions in the final plan are contained in the terminal nodes. Simply read them from left to

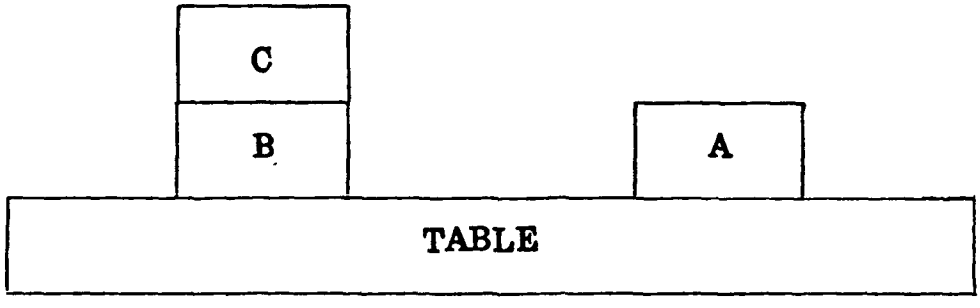


FIGURE 2-1

INITIAL CONFIGURATION FOR BLOCKS WORLD PROBLEM

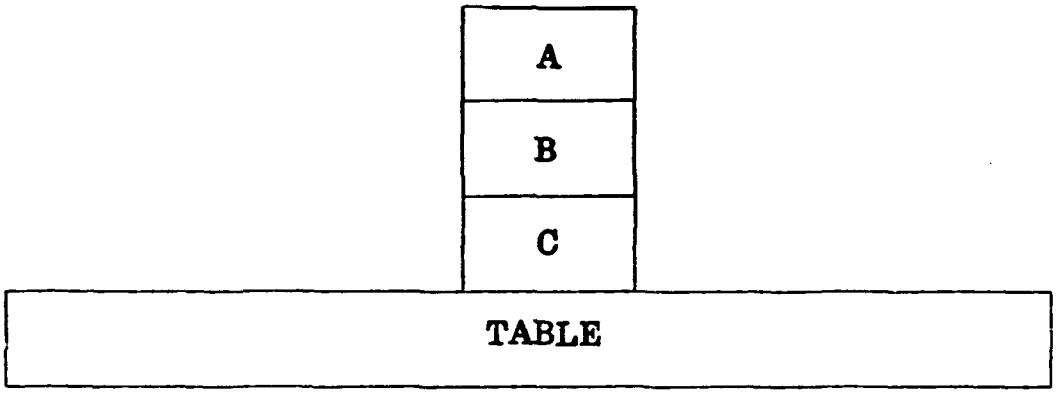


FIGURE 2-2

FINAL CONFIGURATION FOR BLOCKS WORLD PROBLEM

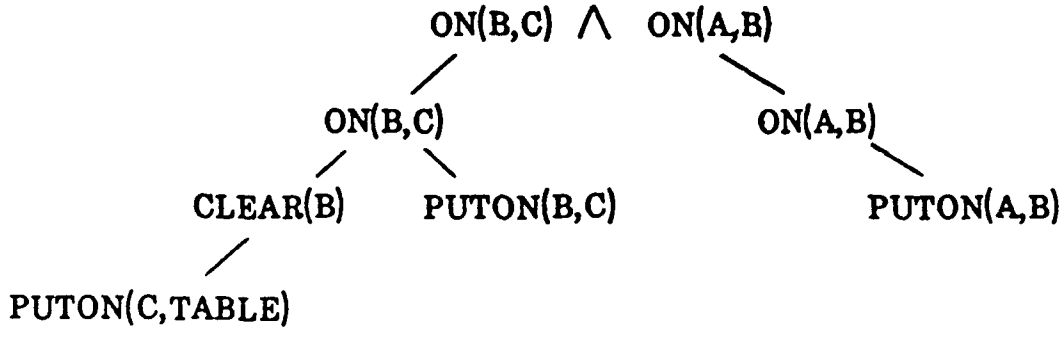


FIGURE 2-3

GOAL TREE FOR BLOCKS WORLD PROBLEM

right. Second, by examining the ancestors for each action, we can determine why each action needs to be taken. For instance, the action PUTON(C,Table) is taken to CLEAR(C), which is necessary before ON(B,C) can be achieved, which is one of the original goals to be achieved. A robot problem solving plan that is represented as a goal tree, therefore, not only consists or the actions that need to be executed, but also of an explanation of why each action is needed. It is also important to note that this goal tree was generated through a recursive 'depth-first' expansion of the tree. For instance, before adding ON(A,B) to the goal tree, all of the descendants of ON(B,C) were added first.

2.1.2 Hierarchical and Parallel Planning

More recent planners (e.g., Sacerdotti, 1976, 1977; Vere, 1981) have introduced two important innovations to this simple goal-seeking process; hierarchical plans and the production of parallel plans. In hierarchical planning, goals and subgoals are, in effect, put in a priority order. The more important goals are first worked out to a reasonable level of detail (i.e., working out the major steps of the plan), and less important goals, the details, are considered later. One common technique for implementing hierarchical planning is to postpone consideration of preconditions. In the above example, for instance, the consideration of the CLEAR(C) precondition would be delayed until ON(A,B) was first considered and added to the goal tree. In another context, say planning a trip from Washington to Chicago, there is little value in determining the details of getting to an airport until the major step of determining which flight to take, and therefore which airport, has been completed.

In parallel planning, instead of trying to achieve a set of goals in some order and then trying to reorder the goals if this doesn't work, the planner assumes all goals can be achieved simultaneously unless interactions between the goals are found that require the imposition of a time order on the consequent actions. For instance, in the above example, it was simply fortuitous that the goal ON(B,C) was explored before ON(A,B). Assume, ON(A,B) were the first goal achieved. Then when ON(B,C) is explored, the first subgoal generated, CLEAR(C), is inconsistent with the fact that ON(A,B) has been achieved, requiring that the planner must undo ON(A,B). In parallel planning, ON(A,B) and ON(B,C) would be assumed to be seperately obtainable goals until it was 'discovered' that a consequence of achieving ON(A,B) is that it violates a precondition of the action PUTON(B,C), namely that CLEAR(B) is true. This would result in the planner requiring that ON(B,C) be attained before ON(A,B).

As can be seen, an important advantage of parallel planning is that it avoids premature commitment to a particular ordering of goals and actions, thereby avoiding the unnecessary work on an infeasible ordering that was initially selected arbitrarily. In order to represent such plans with parallel branches, the goal tree format needs to be expanded so that goals/actions can be represented in parallel, in effect, resulting in hierarchically more detailed PERT-like networks. In AI, such PERT-like networks are usually referred to as a Procedural Nets.

2.1.3 Skeleton Plans

Another approach to planning involves the use of stored

skeleton plans that outline a general sequence of steps for solving a variety of planning problems (Friedland, 1979). Planning with skeleton plans usually proceeds in two stages. First a skeleton plan or subplan is found that is applicable to the given problem. Second, the abstract steps in the plan are filled in with problem-solving operators relevant to the particular problem domain. This instantiation process involves large amounts of domain-specific knowledge, often working through several levels of generality until a problem-solving operator is found to accomplish each step in a skeletal plan. If a suitable instantiation is found for each abstracted step, the plan as a whole will be successful.

2.1.4 Opportunistic Planning

Another approach to planning is that of opportunistic planning (Hayesroth, 1980). This approach is characterized by a flexible control strategy that allows development of a plan in a combined bottom-up and top-down fashion. Problem solving actions will switch from one aspect of the planning problem to another depending on where the next best opportunity exists to further refine a partially completed plan. One result of this planning approach is that the planner will generate various islands of planning action (i.e., local subplans) that may be merged into a single integrated plan near the end of the planning process.

2.1.5 Meta Planning

Meta planning is a general term that refers to reasoning about the planning process. In complex problem domains where alternative problem solving strategies are available, it is useful to incorporate in a planner a mechanism that explicitly reasons about the best

approach to solving a planning problem. Examples of this kind of reasoning can be found in the MOLGEN system (Stifick, 1981) and in PAM (Wilensky, 1981).

2.1.6 Planning in Uncertain Environments

A major restriction of most planning systems is that the planners must have sufficient information about the target world to simulate courses of action to the level of detail required to be able to predict all consequences of each proposed action. Clearly, this complete world knowledge is not available in most practical situations. Indeed, uncertainty about the target world usually comes in many forms, such as a lack of information about the present world state, an inability to fully predict the affect of an action on the world state, or even an inability to anticipate the response of other agents in the target world. Within the action planning area, a number of techniques are being explored for their potential to handle these types of uncertainty. One technique for planning in uncertain environments involves the use of information goals, requiring that certain data be acquired during plan execution. For example, a plan may be developed that includes the action "move object A to location x", even if the location of A is unknown to the planner. This can be done by inserting an information collection goal of "Find location of object A", which can instantiate a "detect-by-TV" operator at the time of execution.

Another set of techniques for planning under uncertainty involves an ability to dynamically repair plans. In most context where there exists a significant degree of uncertainty, it is impossible to plan long sequences of actions in advance, because

unexpected conditions usually occur very early during the execution of the plan. In this type of context the planner must anticipate a need for replanning. An economical way to do this type of replanning is to isolate the previous goals' actions affected by the newly discovered conditions, and then to replace them with new subplans which will work in the new state. In other words, attempt to save as much of the original goal tree as possible. This type of replanning technique can be found in Hayes (1975).

2.1.7 Explicitly Dealing With Time

Most planning systems treat actions as though they occured instantaneously. They incorporate no model of what happens during the execution of an action, and consequently are incapable of describing world states during action execution. Although the need for such time-dependent models is commonly accepted (see Sacerdotti, 1980), there is no ongoing work in this area. (See Hendrix, 1973, for some early work.)

Another way in which time can be dealt with involves the scheduling of actions. In particular, Tate and Daniels (1977) and Vere (1981) describe planners that translates simultaneous actions in parallel plans into specific action start and end times in a PERT chart.

2.1.8 Distributed Execution of Plans

When parallel branches of a plan are generated, it may be desirable to also execute them in parallel. The difficulty that often results with such distributed execution is the occurance of interactions between different execution modules. Typically, interactions are detrimental, and generate problems involving resource

conflicts and the undoing of each others' achievements.

Techniques being developed to resolve these types of problems usually involve communication protocols that keep the various execution modules informed of each others relevant activities, (e.g., Smith 1979) and general procedures that enforce independence between modules by locally resolving interactions, (e.g., such as when two automobiles arrive simultaneously at an intersection and the vehicle on the right side has the right of way).

2.1.9 Interactive Planning

Research in robot and other automatic problem solving domains have resulted in a number of planning systems that function autonomously, but have limited capabilities. One way to expand the capability of such systems is to develop interactive systems where both the human user and the automatic planner cooperate to develop a satisfactory plan. An example of such an interactive planner is found in Wilkins and Robinson (1981).

2.1.10 Relevance to Adversarial Planning

It is clear that the above-discussed techniques and AI planning issues are relevant to any planning domain, including adversarial planning. Unfortunately, from the perspective of planning problems involving an adversary, this work is inherently limited. This is because neither the formalisms for representing plans nor the planning procedures have any convenient mechanisms for incorporating the goals and actions of an adversary.

To illustrate the importance of incorporating an adversary's goals, consider again the work in planning under uncertainty. As noted there, information collection goals are sometimes added to a

plan to collect necessary data at the time of plan execution. When an adversary is present, however, he is likely to have a counterplan of his own to prevent the collection of the required information, and may use any of a variety of tactics (e.g., deception) to achieve this end. Unless the adversary's counter goals and possible actions are explicitly taken into account during planning, the information goal is not likely to be achieved.

A similar problem occurs with the work in distributed planning and execution, where the use of communication protocols and procedures to enforce coordination are relied upon. When an adversary is present, that adversary is likely to engage in actions to disrupt communication and coordination. Consequently, unless these adversarial countergoals are explicitly considered in planning, successful execution of the plan is unlikely.

2.2 AI Game Playing Programs

As discussed above, most of the research in AI planning has been done in the context of planning the actions of a single agent. Although this work addresses a number of issues, it does not directly attack the problem of planning against an adversary. Another area of AI planning research, that is specifically oriented toward planning against an adversary, involves the development of intelligent game playing programs. The relevance of this work to the more general problem of planning against an adversary is discussed below.

2.2.1 Goal-Driven Game Playing

For purposes of this discussion, two types of game playing programs are distinguished as those which utilize a 'planning without

goals' approach and those which utilize a 'planning with goals' approach. A typical example of the 'planning without goals' approach is seen in the Northwestern University chess program (Slate & Atkinson, 1978). This program selects a move by examining all possible sequences for six or seven moves. It then does a weighted linear evaluation of the features (relative number of points, mobility, etc.) of each of the terminal positions. The program then picks the move that maximizes the minimum possible evaluation in the terminal positions. At no point during the planning process does this program focus its attention on achieving specified goals or objectives. Or to put it another way, the program contains no conceptual knowledge about what it should be trying to do, rather it uses the computational power and speed of the computer to review everything it can do and then through a simple decision rule, chooses the 'best'.

This approach has been very successful in games such as checkers and chess where the average number of moves in a position (around 8 for checkers and 25 for chess) is small enough so that a more or less thorough examination of all possible move sequences, for at least several moves ahead, is feasible. In fact, virtually all of the chess and checkers programs noted in the popular literature, such as in computer chess tournaments or programs available for home computers, use some variant of this approach. Unfortunately, in other games, such as the oriental game Go, where there are around 250 legal moves per position, and 'real life' planning under adversity situations, where an opponent has virtually a limitless variety of actions available to him, the use of this type of exhaustive search

approach is not feasible.

An alternative approach to game playing, that is more in the mainstream of AI research, involves the use of goals and objectives to control the move look-ahead process; see Berliner (1977), Pitrat (1976), Wilkins (1979), Reitman and Wilcox (1979), Lehner (1983). The essence of this approach is that if a program is careful in selecting objectives it should try to achieve, then it should be sufficient for planning purposes to only examine actions that potentially aid in achieving those objectives.

To illustrate this goal-oriented approach, assume that we have a chess position where one player, White, believes it feasible to find a mating combination. Consequently, White starts with the goal of CAPTURE_KING. To do this, White isolates a square he needs to occupy with his queen to achieve checkmate. However, this square is protected by a black knight. Consequently, REMOVE_KNIGHT from this square becomes a subgoal. This type of subgoal can be achieved by actually capturing the piece, so CAPTURE_KNIGHT becomes a subgoal of REMOVE_KNIGHT. If the knight can be directly captured, then that capture move becomes the first move. If not, a new subgoal THREATEN_KNIGHT is activated that results in a move that threatens to capture the knight on the next move. In this example, therefore, the first move of a goal-driven look-ahead is one that threatens the knight, so that it can eventually be captured, so that it is no longer defending the black king, so checkmate can be achieved. By carefully considering objectives and goals at each step in look-ahead in this way a program is able to isolate only a few moves that need to be examined.

2.2.2 Goal Pairing: Predicting an Adversary's Goals

Of course, in order to effectively limit the number of possible move sequences to examine, the game playing program must not only determine its own goals, but it must be able to guess the goals of an opponent. Surprisingly this is often not nearly as difficult as might be anticipated. This is because of a general technique (often only implicitly used by developers of AI game playing programs) that will be referred to as 'goal pairing'. Goal pairing is simply the process of identifying an adversary's countergoal to each friendly goal or subgoal that has been generated. In our chess example, for instance, it is reasonable to assume that White's goal of CAPTURE_KING corresponds to Black's goal of SAVE_KING and furthermore that Black would like to prevent the removal of his knight, by preventing its capture. Therefore, after a white move that threatens black's knight, black's move will be one that removes that threat. Figure 2-4 lists the White/Black goal pairs that would result from this example.

It should be noted that most of the research on the use of goals and plans game playing has been oriented toward developing tactical analyses programs. That is, the planners are designed to handle tactical planning problems where even 'minor' deviations from a correct move sequence can lead to failure. The next section discusses some work extending these techniques to high-level strategic planning.

2.2.3 Using Scenarios to Support Strategic Planning

In many adversarial planning situations, in both game playing and real-life conflicts, human planners consider high-level strategic plans that can not be analyzed by the same type of precise action-by-action look-ahead that is characteristic of tactical

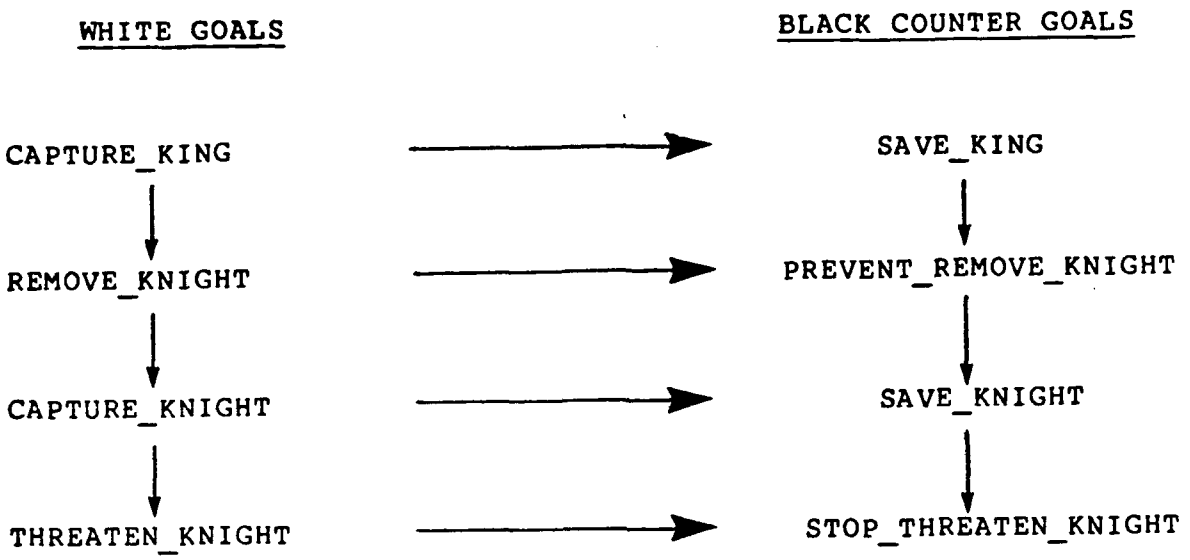


FIGURE 2-4

AN EXAMPLE OF GOAL PAIRING

analysis. This is because the strategic plan is at such a high level that there will usually be a large number of reasonable move sequences that are consistent with high-level strategic goals. Or, put another way, even after goals and objectives are specified for both sides, there are too many reasonable plans of action consistent with those objectives and goals to be able to consider them all. In military planning, for instance, it is not feasible to translate a Corps level concept plan into a precise plan of action where all possible friendly and enemy actions are taken into detailed consideration and anticipated. Despite these problems, it is clear that human planners are able to perform some type of strategic planning or look-ahead that anticipate likely results.

What often makes this type of strategic planning possible is that usually all the action sequences that follow the basic pattern in a strategic plan will have a number of common positional side effects. Or, equivalently, any one sequence of actions that attains a strategic plan is representative, strategically, of all the possible sequences. Consequently, it is possible to determine some of the consequences of a proposed strategic plan by exmaining just one specific plan of action to implement that plan and generalizing results to other possible plans of action. In Lehner (1983), this is referred to as 'representative searching', in military planning it is sometimes referred to as 'playing out a scenario'.

A representative search program for strategic planning in Go is found in Lehner (1983). An example illustrating how this program works is provided below. Note that the spatial/visual nature of strategic plans in Go should make this example understandable for

readers unfamiliar with the game.

The program began with the position in Figure 2-5. The White stone at O3 and O4 are loosely surrounded by the Black stones at K4 and around Q4. Black, the program, starts with the goal of trying to capture these White stones. Consequently, the representative search program starts with a top-level goal of CAPTURE the White stones around O3, and assumes Black's countergoal is to SAVE those stones. In order to capture a group of stones in Go, one must first surround them. Consequently, the first subgoal of CAPTURE (O3) is to SURROUND (O3), for which White's counter goal is to ESCAPE. Now in order to effectively prevent ESCAPE, Black must close up the line between K4 and Q6, by placing stones in between them. Black can do this by playing a sequence of stones from Q6 to K4 (ENCLOSE_RIGHT) or by starting at K4 and moving toward Q6 (ENCLOSE_LEFT). The program starts with the former. This returns the move Ø6. After O6 is played, the only avenue of ESCAPE for White is between the stones K4 and O6, so White's assumed response is M5.

After the White play at M5, attempting to enclose the White stones by playing stones in the K4-O6 line is not feasible because White has already broken through that line. Consequently, a new goal of CREATE_LINE, to continue the attack by creating a new line of attack is called. Also, since the Black stone at K4 is also now loosely surrounded, the Black goal of SAVE (K4) is also added, resulting in an ESCAPE subgoal.

The conjunction of the Black CREATE_LINE and ESCAPE goals results in the move at K6. White continues his own ESCAPE by smashing through the newly formed K6-O6 line with a play at M7, at which point

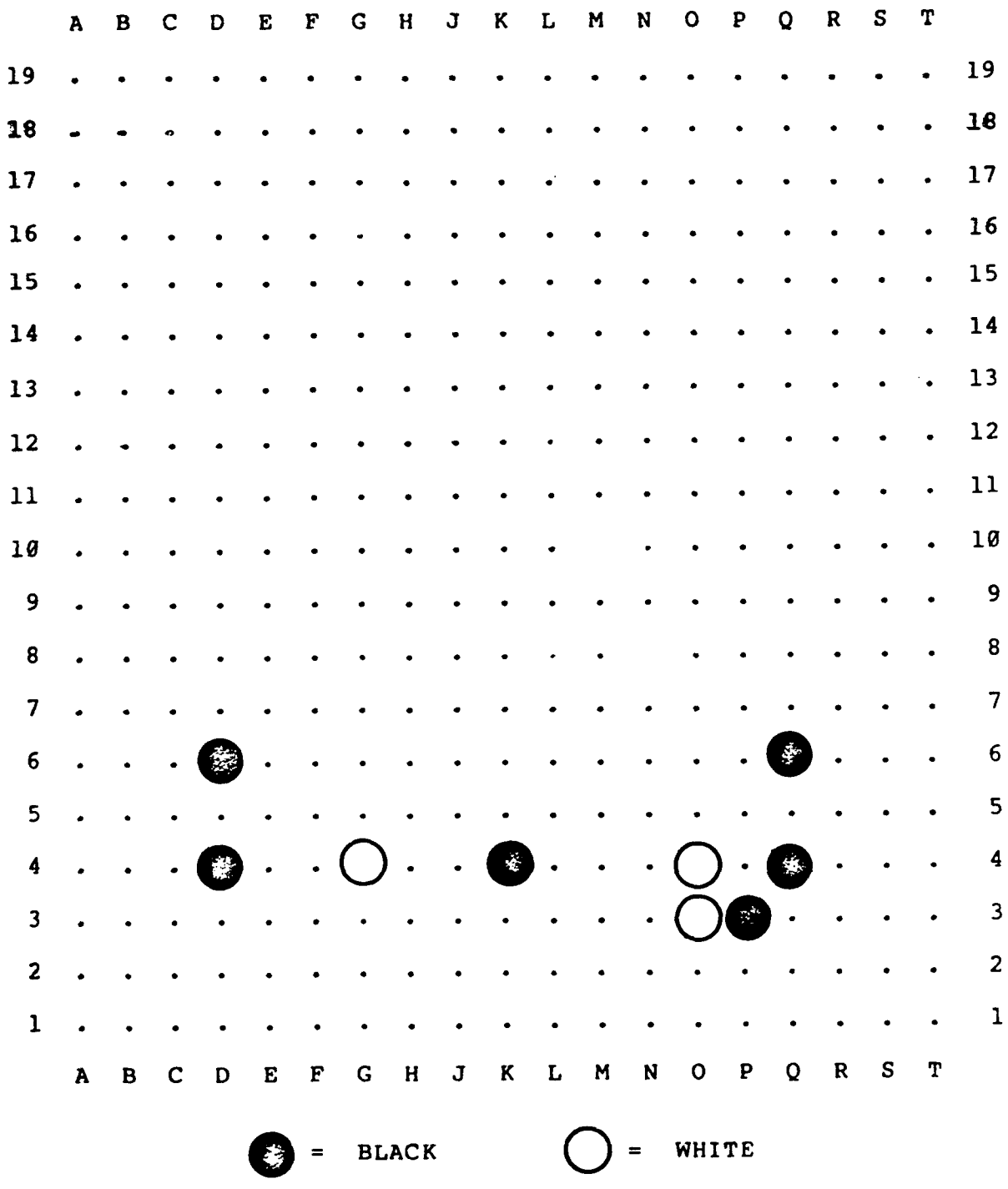


FIGURE 2-5

INITIAL POSITION IN REPRESENTATIVE SEARCH SEQUENCE

White has successfully escaped, because CREATE_LINE cannot build a new attack line. However, also at this point the representative search program 'notices' that in the process of attacking the White stones on the right, he has nearly surrounded the White stone at G4. Consequently, he can now switch direction and attempt to SURROUND (G4).

There are several things to note about this example. First, that any sequence of reasonable moves consistent with the direction of play depicted by with the arrows in Figure 2-6 (of which there are hundreds) would have the same positional consequences, namely that White escapes, but that Black can start a new attack in a different direction. Consequently, other move sequences that attempt to achieve the high-level goal of SURROUNDED (O3) do not need to be considered in detail. Second, the fact that Black's original goal was unobtainable, but that pursuing it set up the opportunity to obtain another, equally valuable goal, is not unique to this example. It is often the case that the primary value in engaging in one strategic course of action is that it directs an adversary's resources and attention in a direction that it makes a second follow-on course of action very feasible. Such indirect strategies and goals are very important to military planning, and are precisely the type of planning which this type representative search or scenario generation procedure should be effective.

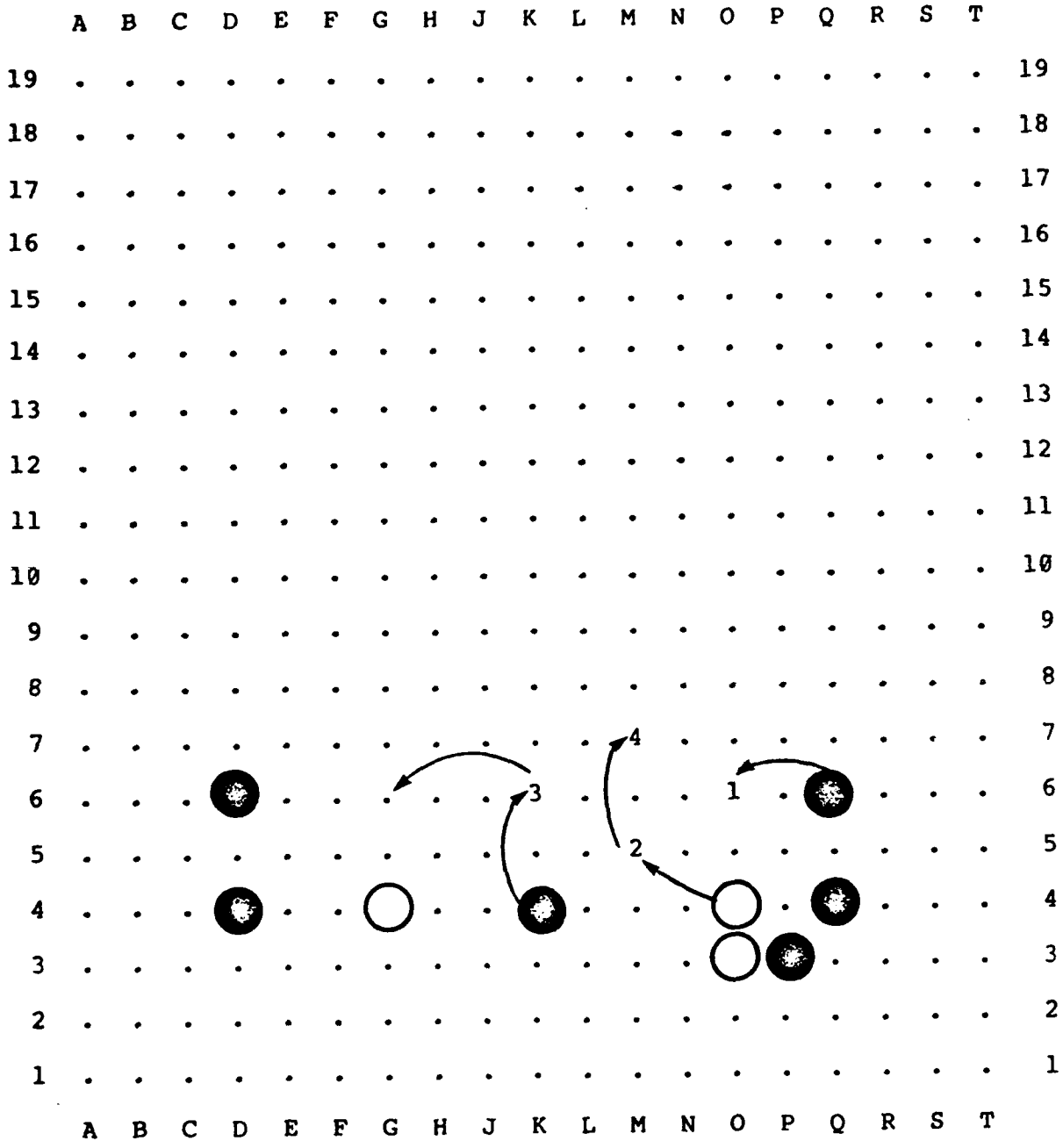


FIGURE 2-6

DIRECTION OF PLAY FOR REPRESENTATIVE SEARCH SEQUENCE

3.0 A THEORY OF ADVERSARIAL PLANNING

Overall the goal of this research program is to extend action planning techniques into domains involving an intelligent adversary. The focus of the first year of this effort has been to implement initial version of an adversarial planner that could serve as a baseline system on top of which alternative techniques for adversarial planning could be implemented and evaluated. Section 3.1 below describes the baseline system, CP/1.0 (Contingency Planner/Version 1.0), which we have developed. Section 3.2 shows an example of CP/1.0 planning behavior in the game of Othello. Section 3.3 then examines each of the individual planning techniques noted in Section 2.0 and discusses how, in theory, they could be embedded within the adversarial CP/x framework.

3.1 Overview of CP/1.0: A Baseline Adversarial Planner

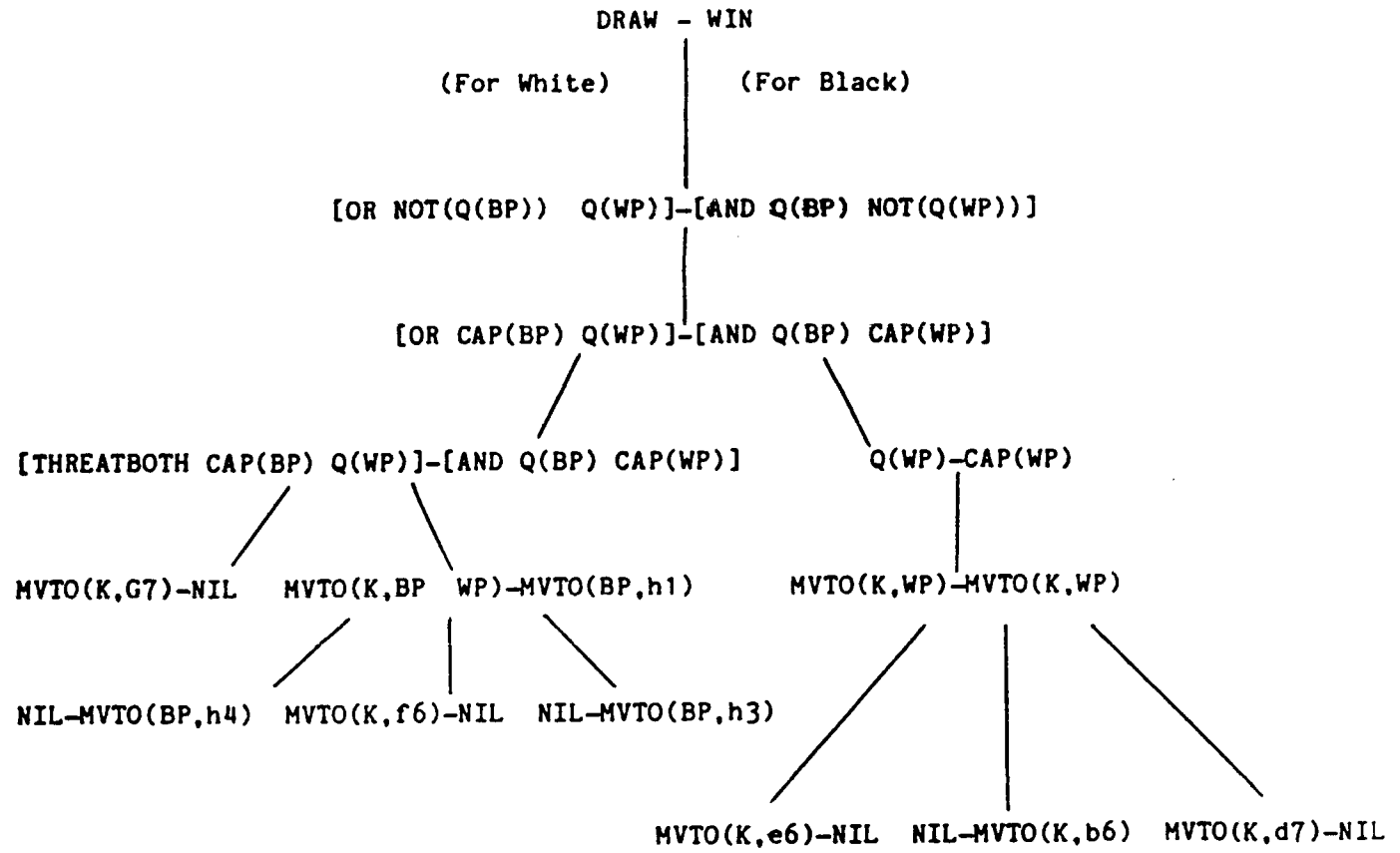
CP/1.0 utilizes two different types of planning procedures: tactical planning and metaplanning. Tactical planning occurs whenever CP/1.0 must determine the outcome of a well-defined goal/countergoal pairing. That is, tactical planning involves the process of generating a move tree to determine whether the planner's goal on the adversary's countergoal will be achieved, and to discover some side effects that result from attempting to achieve a goal. Metaplanning, on the other hand, involves the use of knowledge about planning to control the planning process. Sections 3.1.1 and 3.2.1 below describe each of these planning procedures.

3.1.1 CP/1.Ø Tactical Planning

Like most planners, CP/1.Ø treats planning as a problem in goal tree formation. Described below are the type of goal trees CP/1.Ø generates in performing a tactical search and how it goes about generating them.

Contingency Goal Trees - Within CP/1.Ø, plans are represented using a general formalism called a Contingency Goal Tree (CGT). CGTs are similar to the goal tree used in single agent action planning, with the important difference that each node in the tree include both a planner's goal and an adversary's assumed countergoal. Figure 3-1 shows a possible CGT for a single move sequence leading to a draw for White in the chess position show in Figure 3-2. CP/1.Ø atempts to solve adversarial planning problems by generating multiple CGTs that combine to form a set of contingency plans against each of the reasonable options available to an adversary. In a game, such as chess, the set of terminal nodes in the CGTs combine to form a move tree. CP/1.Ø accepts as input any incomplete CGT and attempts to generate a set of expanded CGTs that represent contingency plans for achieving the top goal in the input CGT. Consequently, CP/1.Ø will either return a set of CGTs; or NIL if the top goal cannot reasonably be achieved. To solve a planning problem, CP/1.Ø uses the following basic procedures for generating CGTs.

Goal-driven, Depth-first CGT Expansion - Given an incomplete CGT, CP/1.Ø will expand the CGT in a recursive depth-first manner. It will begin by identifying the first node in the input CGT that contains a goal pair for which a well-defined action is not defined and will process the goal for the side on the move.



Q( ) = Queen pawn

CAP( ) = CAPTURE

BP = Black Pawn

WP = White Pawn

FIGURE 3-1

ILLUSTRATIVE CONTINGENCY GOAL TREE

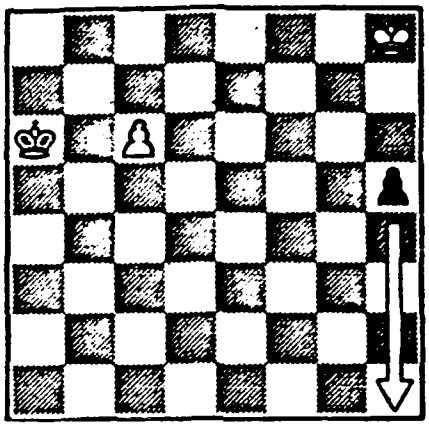


FIGURE 3-2

A STUDY BY RETI. WHITE TO MOVE AND DRAW.

BLACK PAWN THREATENING TO RUN TO H1.

CP/1.0 enters a search problem with a top-level goal-countergoal pairing and a possible side effects list (described below). Depending on the position, processing of the 'present' goal (usually the goal most recently added to the CGT) of the side on the move will result in the generation of a pair of subgoals, which are then added onto the CGT, or it will result in the generation of a move. If a new goal pair is added to the CGT, then the new present goal of the side on the move is processed to produce a new subgoal pair or move. When a move is proposed, it is placed on the hypothetical board. The most recently added goal for the new side on the move becomes the present goal which is processed to see if a new subgoal pair or move can be generated.

Processing of the present goal involves reading in a 'goal object' from a separate file and executing the procedures attached to the goal object. Goal objects are simply structures containing data and functions acting on this data. They interface with the particular problem environment through four functions. 'Subgoal' returns a list of subgoal(s) that is (are) currently most appropriate. 'Countergoal' examines the environment and returns a list of suitable countergoal(s). 'Failed' tests if the goal in the present position has clearly failed. 'Succeeded' tests if the goal in the present position has already succeeded. 'Feasible' quickly tests the feasibility of a goal and returns true if the goal could be feasible in the current environment. 'Action' updates the enviornment as a function of the goal/countergoal pairing.

Processing of an instantiated goal object for the side on move will generate an assumed countergoal for the opposing side, along with

one of the following:

1. an update of the hypothetical world, after which CP/1.0 calls a domain specific utility to determine which side moves next;
2. a single subgoal, which is added to the CGT and is then processed as the new present goal;
3. an incomplete subtree, usually specifying a sequence of subgoals, of which the first subgoal is processed as the new present goal after the entire subtree is added to the CGT;
4. a NIL if processing of the present goal does not result in an action, subgoal or subtree, in which case the antecedent node for the side on the move is once again processed as the new present goal, thereby starting a new branch of the CGT.
5. A 'failed' or 'succeeded', which is similar to No. 4 above except that instead of moving to an antecedent node, a move backup must occur as described in the subsection below.

Goal-driven backup - An individual move sequence terminates whenever either the planner's or adversary's top-level goal is clearly obtainable. Obtaining the top goal, however, does not necessarily imply that the move sequence is a success. A move sequence may result in a number of unanticipated side effects that result in providing the opposing side with new opportunities. For instance, if the planner enters a search problem with a SAVE(X) goal, paired with an adversary's CAPTURE(X) countergoal, a particular sequence may result

in the planner successfully saving X, but the situation has changed such that the adversary now has a new target, Y, to pursue. It is now feasible for the adversary to attempt to CAPTURE(Y) instead. In CP/1.Ø, unanticipated side effects are defined as new opportunities, which in turn are defined as new goals to pursue that were not feasible in the original position.

In order to find new goals, CP/1.Ø engages in a sequence of 'information searches.' At the end of a move sequence where one side achieved its initial goal, CP/1.Ø iterates through a set of recursive calls to CP/1.Ø, with each of the goals in a 'possible side effects' list, to try to find a way to continue play for the opposing side. The type of move backup that will occur depends on whether or not there exist new goals for the opposing side.

Whenever processing of the present goal returns a NIL, and CP/1.Ø cannot move up to an antecedent goal (e.g., when it is the top goal); or when it returns a 'failed', a backup on the proposed move sequence must take place. In CP/1.Ø the procedure for determining how far to backup is goal-driven and proceeds according to the following three steps:

1. iterate through a sequence of information searches on the position, by recursively calling CP/1.Ø, to identify new goals that can be pursued by the failing side;
2. based on the results of the information searches, identify the first node in the CGT (assuming a depth-first ordering of nodes) that is impacted; and

3. remove all nodes that were generated after the impacted node (which automatically includes the moves identified in the terminal nodes), modify the impacted node through a conjunction or disjunction of new goals, and continue the CGT generation process.

The information searches in 1. proceed as described above. Regarding 2. and 3., if the information searches do not discover new goals, then the only node in the CGT that is clearly impacted is the one that directly generated the most recent move for the side that failed. For the side on the move, the goal is modified to be a conjunction of that goal and NOT[MOVE(X)], where MOVE(X) is the move that failed. All nodes added to the CGT after this modified goal are removed and CGT expansion continues as before.

If the information searches do discover unanticipated side effects (i.e., new goals to consider), then it is necessary to discover which nodes in the CGT the new goal interacts with. That is, for each goal in the CGT, it needs to be determined if the discovery of this new opportunity would affect how the planner should go about trying to achieve that goal. Since goal objects carry with them lists of possible side effects, it is a matter of selecting the first goal in the CGT that has the discovered goal on its side effect list. Frequently, it is assumed that a newly discovered goal impacts the top goal in the CGT, in which case all generated nodes in the CGT are removed and the top goal is modified to be a conjunction or disjunction of the original top goal with the new goal. However, in various domains, such as some war games, there may be a number of

independent areas of local conflict, where a discovery of local side effects clearly do not impact overall planning.

3.1.2 Metaplanning in CP/1.0

The metaplanning facility in CP/1.0 is a general, extendable facility for deciding what planning strategies to use. Through the use of domain specific pattern matching routines, it can be used to select a top-level goal pair for a contingency goal tree. It can also be used to select between the tactical planning mechanism described in SXection 3.1.1 and any other planning mechanisms that may be provided. Other planning mechanisms will probably be most useful in the opening moves of a game like Chess, where a number of standard book openings are used, and in the final moves of a game like Othello, where there are few possible moves leading to widely varying outcomes.

The metaplanning facility, like the tactical planning facility, uses a type of goal objects. However, unlike the tactical planner, the goal objects are not paired because the metaplanner does not attempt to match the planning mechanism of the opponent. Like the tactical planner, each metagol object contains a feasibility test. Rather than a list of possible subgoals, the metagoal objects contain a list of planning actions to be taken in sequence to achieve the goal. Examples of types of planning actions are contingency planning using a specific top-level goal pair and exhaustive search. Each metagoal object may also include an alternate metagoal object to be tried if the first fails.

The metaplanner works in a very straightforward manner. When it is called, it is given an initial metagol object to attempt. It first uses the feasibility test to assure that the plan outlined in

the frame is reasonable in the given situation. If it is, the metaplanner attempts to use the first planning action from the planning action list. The metaplanner continues to use the planning actions on the list until the list is exhausted or until one of the planning actions fails. If a planning action fails or the metagoal is not feasible, the metaplanner attempts to process the alternative metagoal frame in the same manner if processed the original.

3.1.3 Summary of CP/1.0

Overall, CP/1.0 implements an approach to tactical planning that incorporates the goal-driven AI game-playing techniques discussed in Section 2.2. In particular, the tactical planning facility incorporates the same techniques found in tactical planners for Chess (Berliner, 1978); Pitrat, 1977; and Wilkins, 1979), and Go (Reitman and Wilcox, 1979). The facility to search for unanticipated side effects can be used to generate the type strategic planning behavior discussed in Lehner (1983). Unfortunately, because of the limited nature of our initial test domain (the game of Othello). This later capability has not been significantly tested.

What is unique about CP/1.0 is not so much what it does, but rather how it does. First, the CP/1.0 software is generic, which makes it suitable for application in multiple domains. Previous systems that have shown adversarial planning capabilities were entirely domain specific, i.e., written as game playing programs. Second CP/1.0 utilizes a formalism for representing plans, CGTs, that is a generalization of the goal tree formalism found in AI action planning. Consequently, both AI action planning and AI game-playing techniques can in theory be incorporated into the CP/x framework. The

CP/x framework therefore provides the opportunity to merge these two sets of techniques into the combined area that we have called adversarial planning.

An example of CP/1.0 planning is found in Appendix A.

3.2 Advanced Adversarial Planning Techniques

The overall goal to this research program is to extend action planning techniques to problems involving planning against an intelligent adversary. Section 2.0 discussed single agent planning techniques that could also be appropriate for adversarial planning problems. Section 3.1 presented our intitial baseline planning system, CP/1.0, that provides a testbed for examining alternative techniques for adversarial planning. In this section each of the *advanced* action planning techniques noted in Section 2.1 are discussed from the perspective of how they might be incorporated within the general CP/x framework.

3.2.1 Goal-Driven Adversarial Planning

As discussed in Section 2.1, action planning techniques are generally goal-driven. Planning is usually treated as a problem in goal tree formation. The same is true of the CP/x approach. In particular, we have extended the concept of a goal tree to that of a contingency goal tree. A CGT is a generalized goal tree in as much as if a CGT did not include the adversary's goal, it would contain the same information as a simple goal tree. In the same way that action planners treat planning as a problem in goal tree formation, CP/1.0 treats adversarial planning as a problem in contingency goal tree formation.

3.2.2 Hierarchical and Parallel Adversarial Planning

As discussed in Section 2.1, hierarchical planning involves a planning process where goals are put in priority order, so that the more important goals, (i.e., the goals that correspond with major steps of a plan) are worked out first and less important goals, (i.e., the details), are considered later. In CP/x framework, hierarchical planning could also occur. In the situation where higher levels of planning abstraction are defined, this would work as follows.

Recall that in CP/1.0 depth first planning, (i.e., subgoal generation) continues until it gets to the point where there exists a utility for updating the hypothetical world for the current goal/countergoal pairing. In the game of Othello, this simply occurred at the level of a move-nil pairing. For a domain in which there exist domain specific utilities for updating the hypothetical world at different levels of goal/countergoal pairs, then CP/x can plan at different levels of abstraction. First, CP/x would plan at the most general level of abstraction by engaging in depth first planning until it reaches the first set of world update utilities. This would result in a set of high-level contingency plans. Then, using these high level contingency goal trees as the input CGTs, CP/x would continue to expand the CGTs by continuing the depth first subgoal generation process beyond the first world update utility encountered and stopping at the second. CP/x would do this recursively for all defined levels of abstraction.

Of course, in order to do this type of hierarchical planning, the domain in which CP/x is applied requires that these update utilites can in fact be defined. In games such as Othello, Chess, and Go,

generating such utilities is not possible. However, in domains such as some war games, utilites may be defined for different levels of conflict (e.g., CORPS, Division, Battalion).

With regard to parallel planning, no explicit techniques have been identified that can incorporate parallel planning into the CP/x adversarial planning environment. However, in the same way that a simple goal tree can be generalized to include parallel branches, it is reasonable to assume that a contingency goal tree can also be generalized to include parallel branches.

3.2.3 Using Skeleton Plans in Adversarial Planning

In action planning, this type of planning involves the use of stored skeletal plans that outline a general sequence of steps for solving problems in a variety of problem areas. Skeletal plans are usually stored in the form of partially complete goal trees. In the case of adversarial planning, skeletal plans can be stored in the form of partially complete CGTs. Since, CP/x accepts as input a partial CGT of any form, (i.e., any part of the tree can be left out) then skeletal planning can occur in a variety of ways. For example, a skeletal plan may have just the top layers of the CGT, with both the friendly and adversary countergoals, defined. On the other hand, a skeletal plan may involve a contingency goal tree with just the friendly goals identified and the adversarial goals left uncertain. In the latter case, then, we would have prestored the major sequence of actions the planner would take and CP/x would try to fill in the adversary's countergoals.

3.2.4 Opportunistic Planning in Adversarial Domains

Opportunistic planning as discussed in Section 2.0 is an

approach to planning that is characterized by flexible planning control structure that allows development of a plan in both bottom-up and top-down manner. At present, CP/x is not compatible with this type of flexible control structure. However, the representative search technique can be emulated within the CP/x approach. As was discussed in Section 2.1, representative searching does exhibit a number of behaviors that are similar to that of opportunistic planning.

3.2.5 Metaplanning in Adversarial Domains

As discussed in Section 3.1, metaplanning is incorporated within the CP/x framework through the use of metagoals. Consequently, metaplanning is one advanced action planning technique that has already been incorporated into the CP/x framework. Unfortunately, because of this limited nature of our test domain (the game of Othello), this facility has not significantly tested.

3.2.6 Adversarial Planning in Uncertain Environments

Adversarial planning in uncertain environments involves at least two dimensions of uncertainty. First, there is the dimension of uncertainty about the adversary's goals. Second, there is the dimension of uncertainty about the position of adversary resources. In the CP/x framework, uncertainty about the opponent's goals and objectives is handled by generating contingency plans for each possible goal and objective. One limit of the existing CP/1.0 framework is that there does not exist a mechanism for limiting the number of possible adversary goals or objectives that are considered. As a result, in some domains there may be a combinatorial explosion of the number of CGTs that are generated. In order to prevent this from

occurring, a deductive mechanism should be added that will identify the most likely adversarial countergoals of each of the friendly goals. Utilization of this deductive mechanism should be goal specific and would be controlled with routines attached to the countergoal slot in the goal objects in a CP/x knowledge base.

For the second problem, where the planner has a lack of knowledge about the position or nature of adversary resources, there exist three subproblems that need to be solved. First, there is the problem of simply deducing or making a best guess as to where and what those resources are. Second, there is the problem of planning to collect information about uncertain resources. Third, there is the problem of generating plans that take into account alternative contingencies in the quantity or location of resources; that is trying to generate plans that are robust against lack of knowledge.

The first problem is purely an inferencing problem and is not within the scope of this research program. The second problem, from our present persepctive, appears to require only the use of information goals, [i.e., COLLECT_INFORMATION(X)] paired with an adversary's countergoals of preventing the collection of information. Whether the adversary does that by directly intercepting our information collection resources or through deception tactics, will be determined by the subgoals available to the adversary's PREVENT_INFORMATION_COLLECTION countergoal.

The third problem is one which we have not yet addressed. At present, the only mechanism that we envision is one that is similar to the contingency goal tree generation process, where instead of having alternative countergoals, alternative possible world states are

considered. A contingency plan for each one would be developed. Although this approach is theoretically feasible, in most cases it will probably lead to a combinatorial explosion in the search space. Consequently, we need to develop a more elegant approach.

3.2.7 Dealing with Time in Adversarial Planning

As discussed in Section 2.1, planning systems that explicitly deal with time are rare. There exist very few if any truly time-dependent models for planning. In the case of adversarial planning, the problem is the same. There do not exist at present what appear to be good models or techniques for incorporating time-dependent outcomes into the adversarial planning process. In the CP/x framework, the best that can be done at the moment, is to attach a time-required-to-generate-outcomes to the world update functions. What that means is that when there exists a local conflict, the utility for updating the world state can also have incorporated within it an estimate as to the time required to resolve that conflict. Given this approach, CP/x can estimate time required to execute one versus another contingency plan. This type of time-dependent world update is reasonable in domains where all actions occur in a linear sequence. However, for parallel planning, which will have parallel-with-respect-to-time actions, it becomes significantly more difficult to use this approach.

3.2.8 Distributed Execution in Adversarial Planning

As discussed in Section 2.1.8, it is sometimes useful in parallel planning to have separate execution modules to execute the actions in each of the parallel branches in a plan. Distributed execution modules need to be able to communicate and coordinate. In

adversarial domains, however, an intelligent adversary will attempt to disrupt this communication and/or coordination. In order to handle that kind of problem within the CP/x framework, we need to incorporate a MAINTAIN_COMMUNICATION and a MAINTAIN_COORDINATION goal and assume that the adversary has a countergoal of DISRUPT_COMMUNICATION and/or DISRUPT COORDINATION. Each of these maintain and disrupt goals then will have subgoals attached to them that correspond to explicit techniques for obtaining these goals.

3.2.9 Interactive Adversarial Planning

There are a number of ways that an adversarial planning system such as CP/x can be adapted to cooperative man/machine planning. One approach that is consistent with some existing interactive action planners, is to modify the goal objects to allow user inputs in the subgoal selection process. The second approach is to let the adversarial planner CP/x play the role of a "devil's advocate", pointing out specific counterplans available to an adversary. Finally, a third, theoretical very interesting approach is one that treats cooperative planning against an adversary as a multi-agent planning problem. In this case, CP/x must explicitly reason about both the adversary's competitive agent's goals and the user's (a cooperative agent) goals. This last approach remains somewhat speculative and we are not sure how to implement it.

3.3.10 Extending CP/x to Multi-Agent Environments

CP/x reasons about a competitive agent's goals and plans through a straightforward process of goal pairing. Namely that every time a subgoal is proposed and added to a goal tree, the adversary's assumed countergoal is immediately generated and also added to the

goal tree. Consequently, wherever CP/x needs to take into account the adversary actions, a complete goal tree representation of the adversary's assumed perspective is also available, leading to immediate selection of possible adversary actions.

Extending this two-agent goal pairing process to a more general multi-agent goal matching process is straightforward. To do this requires primarily that

- CP/x itself be generalized to accept as input the top-level goals of any number of agents, and
- The 'countergoal' selection procedures in the goal definitions needs to be generalized to generate other goals for all agents, both competitive and cooperative.

Although a generalized CP/x provides a framework for processing the goals of any number of cooperative or competitive agents, it does not necessarily imply that the generalied CP/x will be an effective planner in this type of environment. The quality of the planning process still depends on the goal processing procedures defined in the goal objects. Specific techniques for selecting matching goals and subgoals, that are appropriate to the multi-agent environment, need to be embedded in the goal objects.

4.0 SUMMARY

According to the original proposal, the goal of the first year of this effort was to achieve the following milestones:

1. Implement a central planning control system for adversarial planning that includes modules for controling goal generation, updating of hypothetical world states, determining affected goals in a previsouly generated goal tree, and replanning/backup procedures.
2. Implement a rudimentary version of the CP/x environment that includes a general ability to define and develop knowledge bases that CP/x can access.
3. Apply CP/x to at least one problem domain, specifically a simple board game simulating a war environment.

In general, these milestones have been met, although the board game selected for testing CP/1.0 was Othello and not a war game. In fact, selection of an appropriate problem domain turned out to be one of the most difficult problems facing the first year effort. This difficulty occurred because of an inherent conflict between the need for a domain rich enough to effectively test CP/1.0 capabilities and the need to minimize knowledge engineering time so that we could focus our efforts on developing CP/1.0 itself.

Othello is an alternating-play game that does not require a large knowledge base of goals for a reasonable level of play, but has never been programmed with knowledge-based search procedures. Consequently, it provided a good domain for evaluating CP/1.0's search behavior. Most interesting war games, on the other hand, were too complex to

have been a suitable year one test domain. They would have required that a substantial amount of time be dedicated to generating domain specific utilities and knowledge engineering.

Regarding future work, the focus for the next two years will be on systematically implementing and testing the various advanced planning techniques discussed in Section 3.2. The immediate focus will be on:

- testing the representative search and metaplanning capabilities that are embedded in CP/1.Ø, but could not be satisfactorily tested in the domain of Othello;
- implementing a facility for planning in domains that involve uncertainty, information goals, and the use of deception tactics; and
- implementing an automated hierarchical planning capability.

REFERENCES

Berliner, H., Chess as problem solving: The development of a tactics analyzer, Unpublished doctoral thesis, Carnegie-Mellon University, 1974.

Fikes, R.E., Hart, P.E., and Nilsson, N.J., "Learning and Executing Generalized Robot Plans," Artificial Intelligence (3), 251-288.

Friedland, P.E., 1979. Knowledge-based experiment design in molecular genetics. Rep. No. 79-771, Computer Science Dept., Stanford University (Doctoral dissertation.)

Hayes, P.J., "A Representation for Robot Plans," Proceedings of the Fourth International Joint Conference on Artificial Intelligenct," Tbilisi USSR, 1975, 181.

Hayes-Roth, B. 1980. Human planning processes. Rep. No. R-2670-ONR, Rand Corp., Santa Monica, California.

Hendrix, G.G., "Modeling Simultaneous Actions and Continuous Processes, Artificial Intelligence, 1973, (4), 145-180.

Lehner, P.E., "Strategic Planning in Go," To appear in book on AI Game Playing, ed. Max Bramer, England: Ellis Horwood Ltd., 1982.

Lehner, P.E. and Patterson, J.F., "Decision Analysis and Artificial Intelligence: Applications to Senior Battle Staff Decisions," Conference Report, Decisions and Designs, Inc., 1981.

Pitrat, J.A., "A chess combination program which uses plans," Artificial Intelligence, 1977, (8), 275-321.

Reitman, W. and Wilcox, B., "Modeling tactical analysis and problem solving in Go," Proceeding of the Tenth Annual Pittsburgh Conference on Modeling and Simulation, 1979, 2133, 2148.

Sacerdoti, E.D., "Planning in a Hierarchy of Abstraction Spaces," Artificial Intelligence, 1974, (5), 115-135.

Sacerdoti, E.D., "Plan Generation and Execution for Robotics," Stanford Research Institute Technical Note 209, 1980.

Slate, D. and Atkins, L., "CHESS 4.5 - The Northwestern University chess program," In Chess Skill in Man and Machine, Frey, P. (ed.), Springer-Verlag, 1977.

Smith, R.G., "A Framework for Distributed Problem Solving," Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan, 1979.

Stefik, M., "Planning with Constraints (MOLGEN: Part 1",) Artificial Intelligence, Vol. 16, No. 2, pp. 111-39, 1981a.

Stefik, M., "Planning and Meta-Planning (MOLGEN: Part 2",) Artificial Intelligence, Vol. 16, No. 2, pp. 141-169, 1981b.

Tate, A. "Generating Project Networks," Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Mass., 1977, 888-893.

Vere, S.A., "Planning in time: Windows and Durations for Activities and Goals," Jet Propulsion Laboratory Report, Pasadena, California, 1981.

Wilensky, R., "PAM," In Inside Computer Understanding , R.C. Schank & C.K. Riesbeck (Eds.), Erlbaum, Hillsdale, N.J., 1981.

Wilkins, D., Using patterns and plans to solve problems and control search. Unpublished doctoral thesis, Stanford University, 1979.

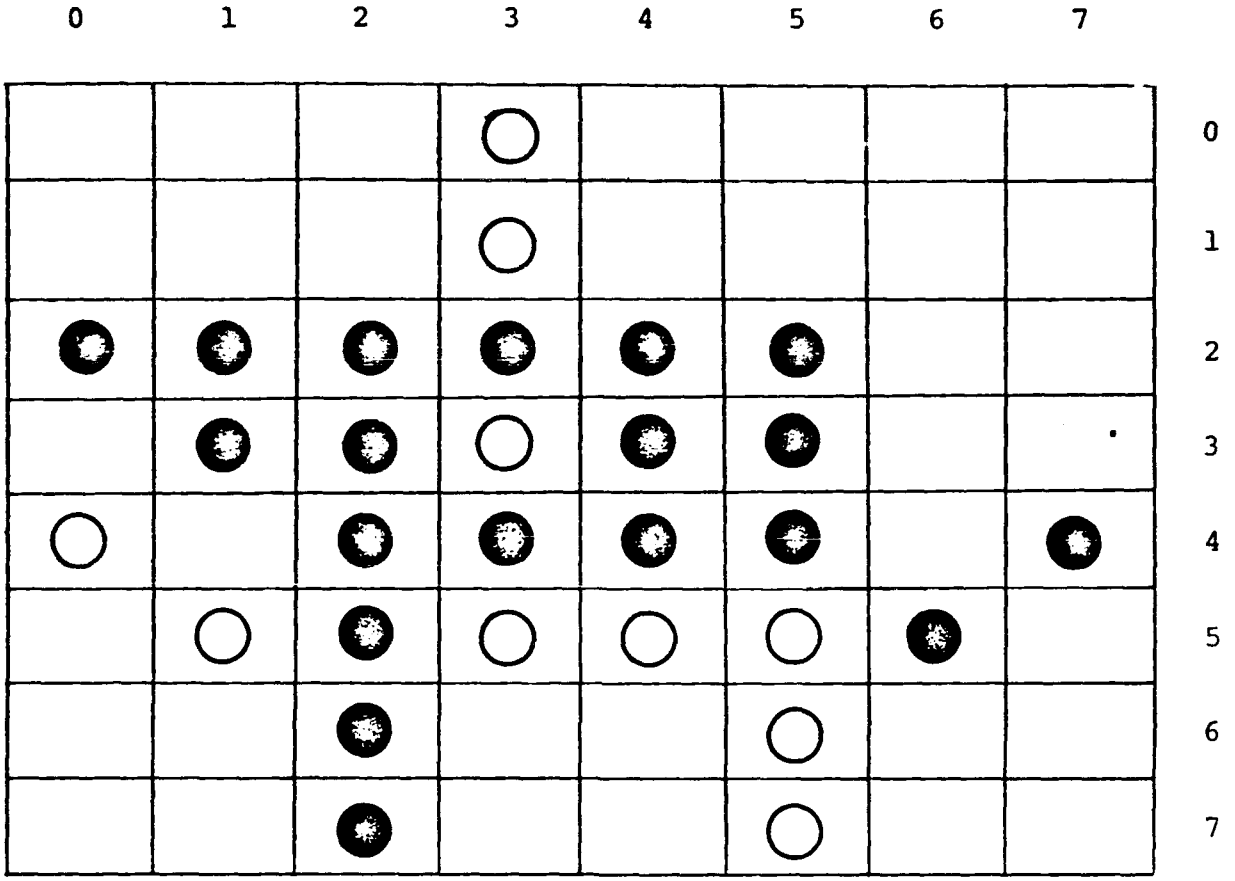
Wilkins, D. and Robinson, A.E., "An Iterative Planning System," Stanford Reserch Institute Technical Note 245, 1981.

APPENDIX A

AN EXAMPLE OF CP/1.Ø TACTICAL SEARCH IN OTHELLO

A.Ø AN EXAMPLE OF CP/1.Ø TACTICAL SEARCH IN OTHELLO

For the board position shown in Figure A-1, CP/1.Ø generated the move tree shown in Figure A-2. Figures A-3 to A-7 show the contingency goal trees associated with each branch of this move tree. The specific goal objects that generated this search are shown in Appendix B.



(0 7) = UPPER RIGHT CORNER (7 0) = LOWER LEFT CORNER

FIGURE A-1

INITIAL POSITION IN CP/1.Ø OTHELLO SEARCH EXAMPLE

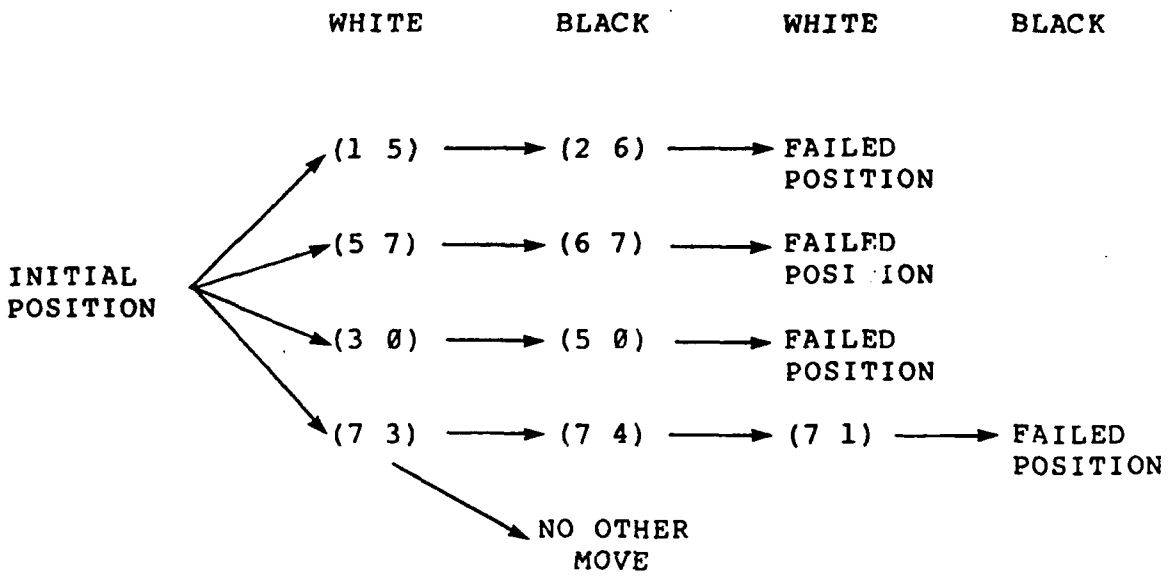


FIGURE A-2

MOVE TREE FOR CP/1.0 OTHELLO SEARCH EXAMPLE

White Moves First

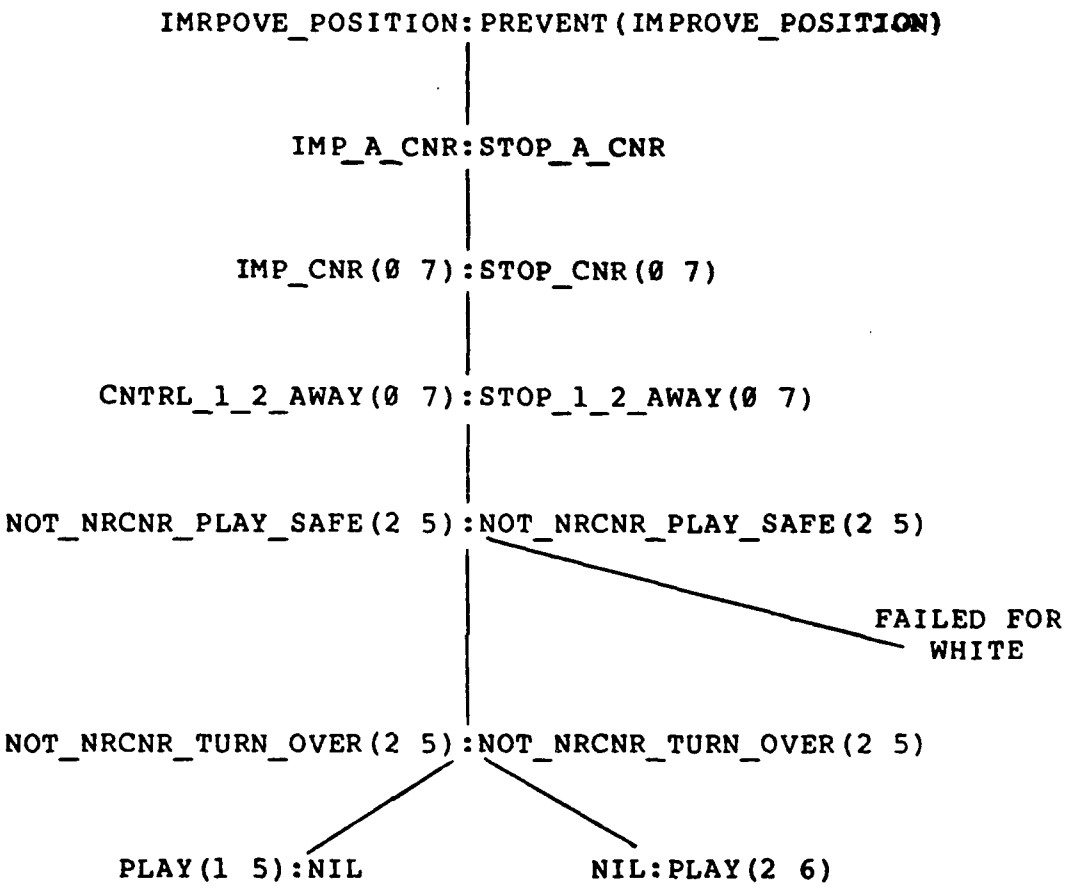
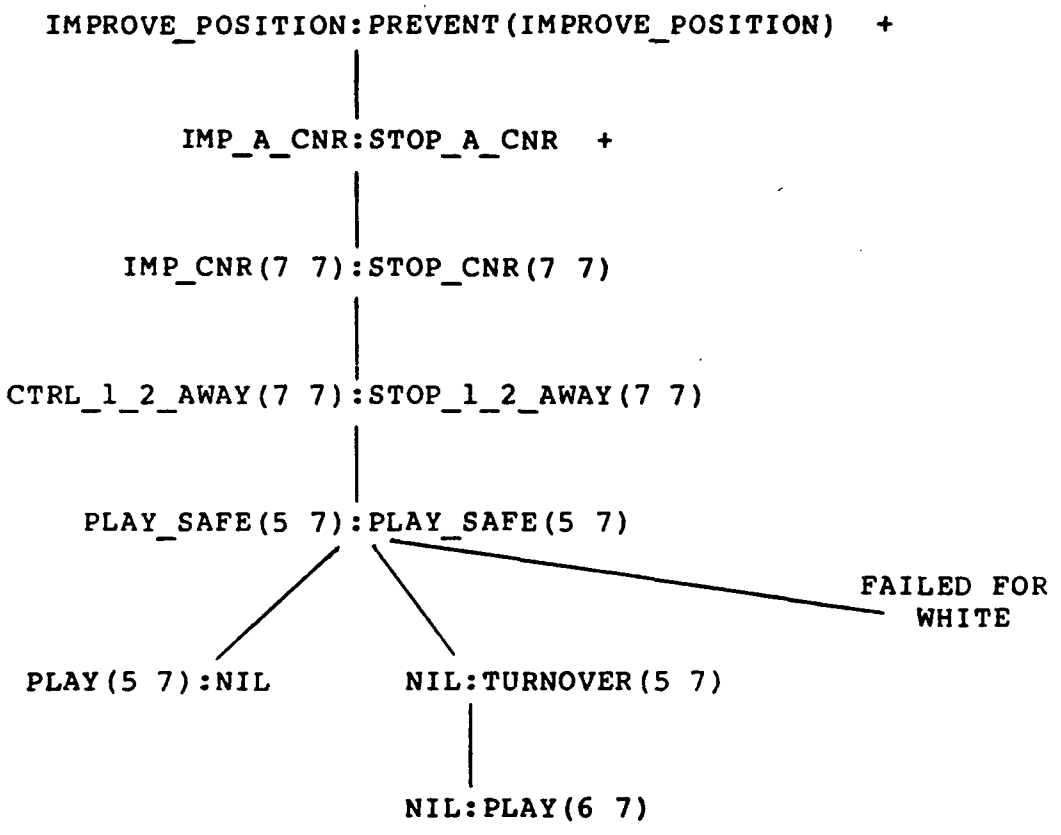


FIGURE A-3

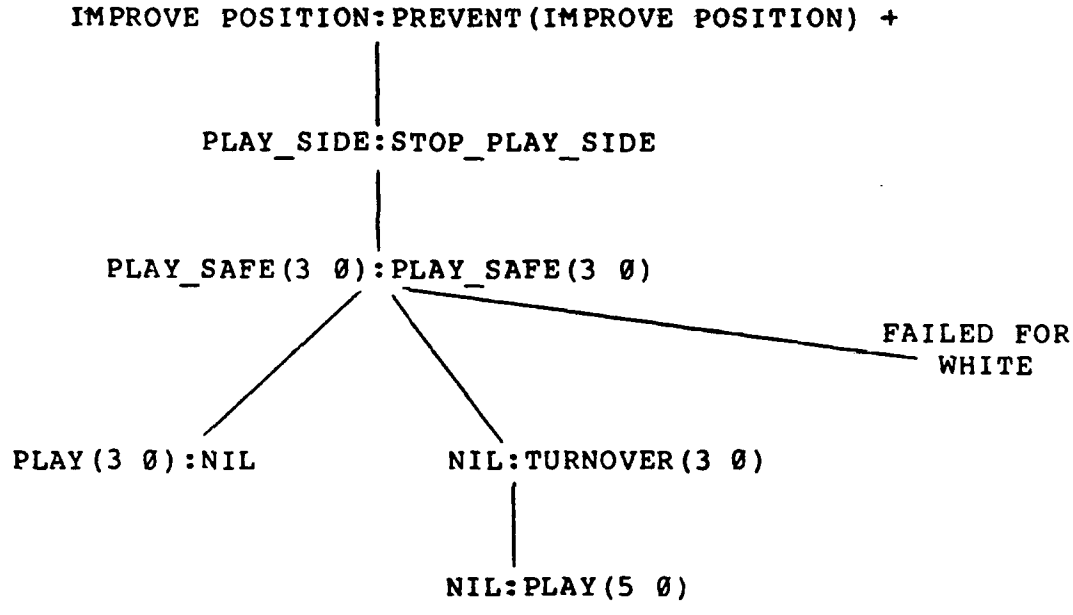
FIRST CONTINGENCY GOAL TREE IN SEARCH



(+ indicates goal-pair kept from previous tree)

FIGURE A-4

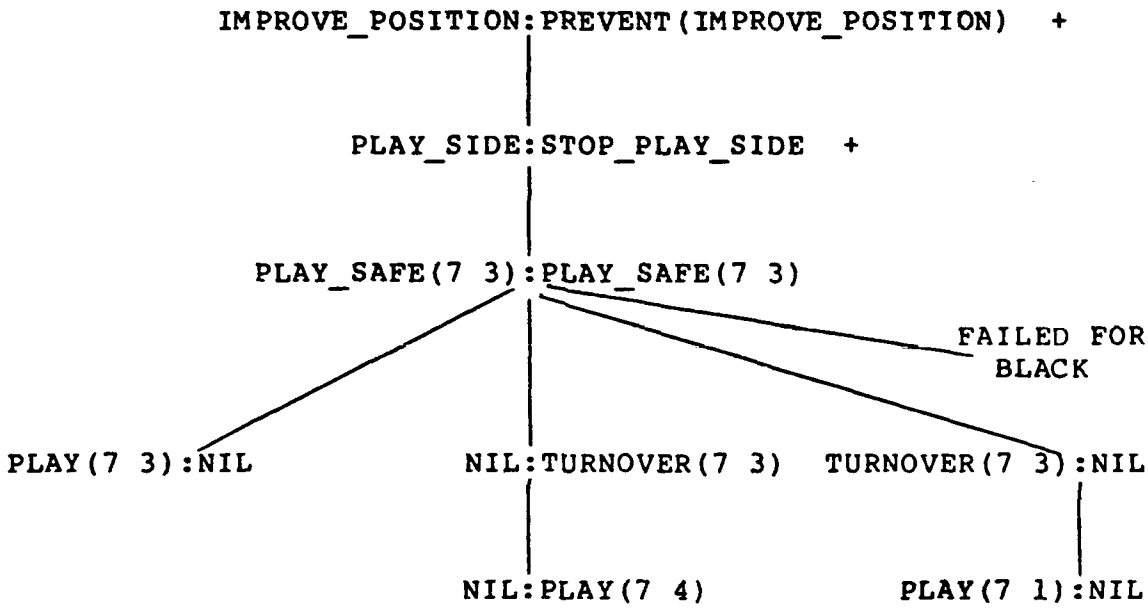
SECOND CONTINGENCY GOAL TREE IN SEARCH



(+ indicates goal-pair kept from previous tree)

FIGURE A-5

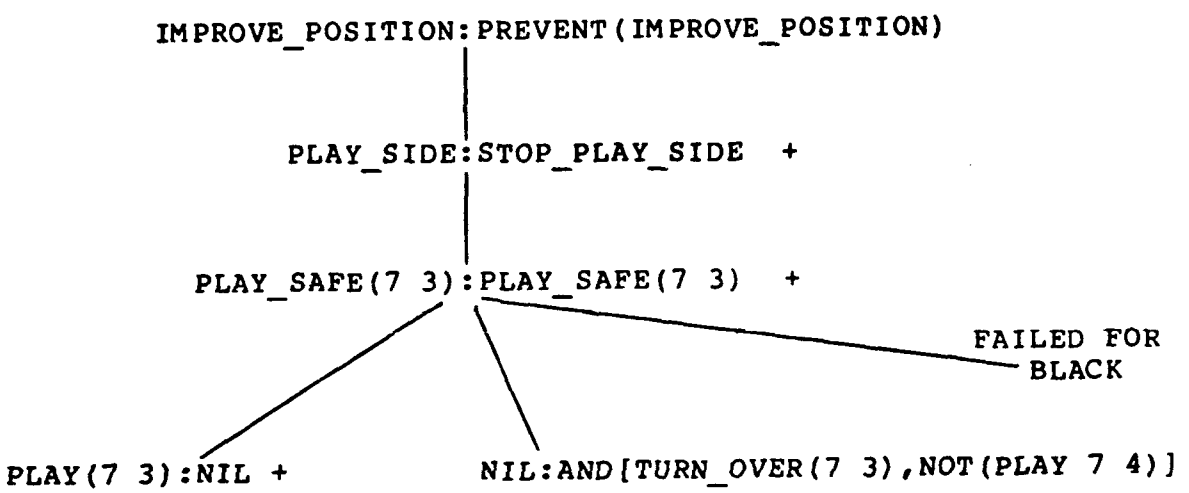
THIRD CONTINGENCY GOAL TREE IN SEARCH



(+ indicates goal-pair kept from previous tree)

FIGURE A-6

FOURTH CONTINGENCY GOAL TREE GENERATED DURING SEARCH



(+ indicates goal-pair kept from previous tree)

FIGURE A-7

FIFTH CONTINGENCY GOAL TREE IN SEARCH

APPENDIX B

SOURCE LISTINGS FOR CP/1.Ø AND OTHELLO

B.Ø SOURCE LISTINGS FOR CP/1.Ø AND OTHELLO

The source listing for CP/1.Ø, the utilities for playing Othello, and the Othello goal objects are provided below. Since CP/1.Ø is an intermediate product that will be significantly enhanced and modified during the next two years of this program, and Othello is primarily a test domain for debugging the CP/1.Ø software, no attempt has been made to provide extensive internal commenting of the code.

Instructions for playing Othello, using CP/1.Ø are found in the Intro.Othello file on the magnetic tape delivered with this report.

Nov 29 13:51 1984 cp1.0 Page 1

```
(def search
  (nlambda (params)
          (prog (selist $rejs tree environment $sorf messflag)
                (cond ((null (cadr params))
                       (setq tree (maketree (tupsualpair))))
                      ((t
                       (setq tree (eval (cadr params)))))
                (setq $rejs (list (list (gensym) (gensym))))
                retry
                (setq tree (ainback tree))
                (cond ((car (lastpair tree))
                       (cond ((sideffects pselist)
                              (go retry)))))
                (switchsides 'goal)
                (return tree))))

(def ainback
  (lambda (tree)
          (prog (newtree environment)
                loop
                (setq $sorf nil)
                (setq newtree (far_as_possible tree))
                (cond ((and (or (null (cur (lastpair tree)))
                                (eq $sorf 'failed))
                            (can_trim newtree))
                       (setq $rejs (cddr $rejs))
                       (setq tree (trim newtree))
                       (switchsides)
                       (go loop))
                      ((and (eq $sorf 'succeeded)
                            (can_trim1 newtree))
                       (setq $rejs (cdr $rejs))
                       (setq tree (trim1 newtree))
                       (go loop)))
                (return newtree))))

(def far_as_possible
  (lambda (tree)
          (cpy environment board)
          (prog (new_tree)
                (switchsides 'countergoal)
                (playout tree)
                (prog nil
                      loop
                      (switchsides)
                      (setq new_tree (complestmove tree))
                      (cond (new_tree
                             (setq tree new_tree)))
                      (and (or (eq $sorf 'succeeded)
                               (null new_tree))
                           (return))
                      (go loop))
                (return tree))))

(def complestmove
  (lambda (tree)
```

```
        (prog nil
              loop
              (or (continue_play)
                  (return tree))
              (setq tree (improve_tree tree))
              (eval (seti (cur (lastpair tree))
                          (quote action)))
              (go loop))))

(def improve_tree
  (lambda (tree)
        (cond ((and (greaterp (length tree) 2)
                    (cond ((all_or_nil (butlast tree)
                                       (listtree
                                        (improve_tree (lastn tree)))))
                          (t
                           (reprop (currejs $rejs) (cur (lastpair tree)))
                           nil))))
              ((null $sorf)
               (expand_tree tree)))))

(def expand_tree
  (lambda (tree)
        (cond ((and (seti (cur (top tree)) (quote action))
                    (eval (seti (cur (top tree)) 'feasible)))
               (setq $sorf nil)
               (setq $rejs (cons (list (sensym) (sensym)) $rejs))
               tree)
              (t
               (cond ((eval (seti (cur (top tree)) 'succeeded))
                      (setq $sorf 'succeeded)
                      tree)
                     ((eval (seti (cur (top tree)) 'failed))
                      (setq $sorf 'failed)
                      nil)
                     (t
                      (setq $have (all_or_nil tree
                                              (select_sub (top tree))))
                      (cond ($have
                             (append (list 'tree
                                           (list (car (top tree))
                                                 (lastn $have)))
                                     (butlast (cddr $have))))))))))))

(def select_sub
  (lambda (parm)
        (prog (current curlist res)
              (setq curlist (set_sub (eval (seti (cur parm) 'subgoal))
                                     (set (currejs $rejs) (cur parm))))
              (setq curlist (subset '(lambda(k)
                                            (eval(seti k 'feasible)))
                                    curlist))
              loop
              (setq current (car curlist))
              (setq curlist (cdr curlist))
              (putprop (currejs $rejs)
```

```
                    (cons current (set (cdr rejs) (cdr rparm)))
                    (cdr rparm))
              (cond ((and current
                          (null (setq res
                                      (expand_tree
                                       (asketree (counterpair current
                                                              (count rparm)))))))
                     (go loop)))
              (return (cond (res
                             (list res nil))
                            (t
                             (list res))))))

(def all_or_nil
  (lambda (list1 list2)
          (cond ((and (and list1 list2)
                      (not (equal list2 (quote (nil)))))
                 (append list1 list2)))))

(def asketree
  (lexpr (npars)
         (list 'tree (list (arg 1)
                           (cond ((greaterp npars 1)
                                  (arg 2)))))))

(def lastpair
  (lambda (tree)
          (cond ((equal (car tree) (quote tree))
                 (lastpair (lastn tree)))
                (t
                 tree))))

(def butlast
  (lambda (list)
          (cond ((or (null list)
                     (eq (length list) 1))
                 nil)
                (t
                 (cons (car list)
                       (butlast (cdr list)))))))

(def sideffects
  (lambda (rselist)
          (setq xxxx nil)
          (princ "searching for side effects")
          (terpri)
          (subset (quote (lambda (k)
                                 (possible (asketree (counterpair k nil)))))
                  rselist)))

(def possible
  (lambda (tree)
          (prog (new_tree)
                (switchsides 'goal)
                (prog nil
                      loop
```

```
                    (switchsides)
                    (setq new_tree (coarleataove tree))
                    (cond (new_tree
                           (setq tree new_tree)))
                    (and (null new_tree)
                         (return))
                    (su loop))
                  (cond ((eq $side 'goal)
                         (return tree))
                        (t
                         nil)))))

(def switchsides
  (lexpr (npar)
         (cond ((or(and(eq npar 1)
                       (eq (ars 1) 'goal))
                   (and(not(eq npar 1))
                       (equal $side 'countergoal)))
                (def cur
                     (lambda (parae)
                             (caar parae)))
                (def count
                     (lambda (parae)
                             (cadar parae)))
                (def curreJs
                     (lambda (list)
                             (caar list)))
                (def addreJs
                     (lambda (reJect)
                             (setq $reJs
                                   (list (list (cons reJect
                                                     (caar $reJs))
                                               (cadar $reJs))
                                         (cadr $reJs)))))
                (def counterpair
                     (lambda (subgoal cparent)
                             (list subgual
                                   (countergoal subgoal cparent))))
                (setq curside (quote white))
                (setq $side 'goal))
               (t
                (def cur
                     (lambda (parae)
                             (cadar parae)))
                (def count
                     (lambda (parae)
                             (caar parae)))
                (def curreJs
                     (lambda (list)
                             (cadar list)))
                (def addreJs
                     (lambda (reJect)
                             (setq $reJs
                                   (list (list (caar $reJs)
                                               (cons reJect
                                                     (cadar $reJs)))
```

```
                                        (cadr $rejs)))))
                (def counterpair
                    (lambda (subgoal cparent)
                            (list (countergoal subgoal cparent)
                                  subgoal)))
                (setq curside (quote black))
                (setq $side 'countergoal)))))

(def countergoal
  (lambda (sub cparent)
          (car (int (eval (seti sub 'countergoal))
                    (eval (seti cparent 'sub_not_on_move))))))

(def int
  (lambda (s1 s2)
          (cond ((and s1 s2)
                 (apply 'append (mapcar '(lambda (k)
                                                 (listcar (mema k s1)))
                                        s2)))
                (s1)
                (s2))))

(def listcar
  (lambda (list)
          (cond ((null list)
                 nil)
                (t
                 (list (car list))))))

(def makegoal
  (macro (call)
         (putprop (cadr call) 't 'feasible)
         (cond (( not (memq (cadr call) targoals))
                (setq targoals (cons (cadr call) targoals))))
         (prog (goalname cur list)
               (setq goalname (cadr call) cur (caddr call) list (cdddr call))
               (prog nil
                     loop
                     (or cur (return))
                     (puton goalname cur)
                     (setq cur (car list))
                     (setq list (cdr list))
                     (go loop))
               (return nil))
         (list (quote putprop)
               (list (quote quote)
                     (cadr call))
               (quote (quote goal))
               (quote (quote type)))))

(def puton
  (macro (call)
         (list (quote putprop)
               (cadr call)
               (list (quote quote)
                     (cadr (eval (caddr call)))))
```

```
            (list (quote quote)
                  (car (eval (caddr call)))))))

(def seti
  (lambda (ata prop)
          (cond ((null ata)
                 nil)
                (t
                 (cond ((and (not (memq ata perasoals))
                             (not (memq ata tempsoals)))
                        (instan ata)))
                 (set ata prop)))))

(def sener
  (lambda (ata)
          (cond ((set ata (quote sener)))
                (t
                 (putprop ata
                          (implode (toslash (explode ata)))
                          (quote sener))))))

(def specf
  (lambda (ata)
          (cond ((set ata (quote specif)))
                (t
                 (putprop ata
                          (implode (makist (fromslash (explode ata))))
                          (quote specif))))))

(def makeform
  (macro (call)
         (list (quote putprop)
               (list (quote quote)
                     (cadr call))
               (list (quote quote)
                     (cddr call))
               (quote (quote form)))))

(def instan
  (macro (call)
         (and (null (set (sener (eval (cadr call))) 'form))
              (princ 'undefined form'))
         (setq tempsoals
               (cons (eval (cadr call))
                     tempsoals))
         (append (list (quote makesoal)
                       (eval (cadr call)))
                 (substit (specf (eval (cadr call)))
                          (set (sener (eval (cadr call)))
                               (quote form))))))

(def $rep
  (lambda (item l tok)
          (mapcar (quote (lambda (k)
                                 (cond ((equal k tok)
                                        item)
```

```
                (cond ((and k
                            (car k))
                       (setq temp
                             (implode (append (explode sen)
                                              (cons 'unmklst (explode k)))))
                       (putprop temp sen 'quote server))
                       (putprop temp k 'quote specif))
                       (return temp))
                      (t
                       (return nil))))))

(def firstpair
  (lambda (tree)
          (cond ((not (equal (car tree)
                             'quote tree))))
                ((caddr tree)
                 (firstpair (caddr tree)))
                (t
                 (top tree)))))

(def listtree
  (lambda (tree)
          (cond (tree
                 (list tree)))))

(def top
  (lambda (tree)
          (cadr tree)))

(def newout
  (lambda (tree)
          (princ 'achieve ')
          (newout2 tree)))

(def newout2
  (lambda (tree)
          (princ (cur (top tree)))
          (cond ((count (top tree))
                 (princ ' without al' wins ')
                 (princ (count (top tree)))))
          (terpri)
          (cond ((greaterp (length tree) 2)
                 (princ 'by ')
                 (newout2 (lastn tree))))))

(def singlethread
  (lambda (tree)
          (and (equal (car tree)
                      'quote tree))
               (or (equal (length tree) 2)
                   (and (equal (length tree) 3)
                        (singlethread (lastn tree)))))))
(def rem_1_branch
  (lambda (tree)
          (cond ((or (eq (length tree) 2)
                     (null tree))
```

```
                nil)
               ((not (singlethread (lastn tree)))
                (append (butlast tree)
                        (list (rem_1_branch (lastn tree)))))
               (t
                (butlast tree)))))

(def can_trim
  (lambda (tree)
          (prog nil
                loop
                (cond ((and (dead (lastpair tree))
                            tree)
                       (setq tree (rem_1_branch tree))
                       (go loop))))
          (cond ((or (singlethread tree)
                     (null tree))
                 nil)
                ((singlethread (lastn tree))
                 (can_trim1 (butlast tree)))
                (t
                 (can_trim (lastn tree))))))
(def trim
  (lambda (tree)
          (prog nil
                loop
                (cond ((and (dead (lastpair tree))
                            tree)
                       (setq tree (rem_1_branch tree))
                       (go loop))))
          (cond ((not (singlethread (lastn tree)))
                 (append (butlast tree)
                         (list (trim (lastn tree)))))
                ((null tree)
                 nil)
                (t
                 (trim1 (butlast tree))))))

(def can_trim1
  (lambda (tree)
          (prog nil
                loop
                (cond ((and tree
                            (dead (lastpair tree)))
                       (setq tree (rem_1_branch tree))
                       (go loop))))
          (greaterp (length tree) 2)))

(def trim1
  (lambda (tree)
          (prog nil
                loop
                (cond ((and (dead (lastpair tree))
                            tree)
                       (setq tree (rem_1_branch tree))
                       (go loop))))
```

```
        (cond ((and (eq (length tree) 3)
                    (eq (length (caddr tree)) 2))
               (cond ((null (caaadaddr tree))
                      (rempror (cureejs $rejs) (cadaadr tree))
                      (maketree (list (caaadr tree)
                                      (goalinst 'andnot
                                                (list (cadaadr tree)
                                                      (cadaadaddr tree))))))
                     (t
                      (rempror (cureejs $rejs) (caaadr tree))
                      (maketree (list (goalinst 'andnot
                                                (list (caaadr tree)
                                                      (caaadaddr tree)))
                                      (cadaadr tree))))))
              ((eq (length tree) 2)
               '(tree ((nil nil) nil)))
              (t
               (append (butlast tree)
                       (list (trim1 (lastn tree))))))))

(def dead
  (lambda (pair)
          (not (or (geti (cur pair) 'action)
                   (geti (count pair) 'action)))))

(def playout
  (lambda (tree)
          (cond ((greaterp (length tree) 2)
                 (mapc (quote playout)
                       (cddr tree)))
                ((lessp (length tree) 3)
                 (cond ((null (cadar (top tree)))
                        (eval (geti (caar (top tree)) 'action))
                        (switchsides 'countergoal))
                       ((null (caar (top tree)))
                        (eval (geti (cadar (top tree)) 'action))
                        (switchsides 'goal))))
                (t
                 (playout (caddr tree))
                 (playout (cdddr tree))))))

(def playboard
  (lambda (tree)
          (cond ((eq (length tree) 2)
                 (cond ((geti (caaadr tree) 'updateaction)
                        (eval (geti (caaadr tree) 'updateaction))
                        t)))
                (t
                 (some 'playboard (cddr tree))))))
(def $aklst
  (lambda (list)
          (append (cons '(( (mapcar (quote (lambda (k)
                                                   (cond (( equal k '!!!) '( ))
                                                         (t
                                                          k))))
                                    list))
```

```
                    (quote (1)1)))))))

(def $unmklst
  (lambda (list)
          (mapcar (quote (lambda (k)
                                 (cond ((memq k (quote ((( ) )) )))
                                       '1/1)
                                       ((equal '| | k)
                                        '|!|)
                                       (t
                                        k))))
                  list)))

(def substit
  (lambda (list form)
          (prog (temp)
                (setq temp (cdr form))
                (mapc (quote (lambda (tic tac)
                                     (setq temp
                                           ($rep tic temp tac))))
                      list
                      (car form))
                (return temp))))

(def subset
  (lambda (funct list)
          (do ((cur (car list)(car list))
               (s))
              ((null list)
               s)
              (setq list (cdr list))
              (cond ((apply funct (list cur))
                     (setq s (append1 s cur)))))))

(def every
  (lambda (funct list)
          (do ((cur (car list) (car list)))
              ((or (null list)
                   (not (eval (list funct 'cur))))
               (cond ((null list)
                      t)))
              (setq list (cdr list)))))

(def some
  (lambda (funct list)
          (do ((cur (car list) (car list)))
              ((or (null list)
                   (eval (list funct 'cur)))
               (cond (list
                      list)))
              (setq list (cdr list)))))

(def  set_sub
  (lambda (set1 set2)
          (mapc '(lambda (k)
                         (setq set1 (allbut k set1)))
```

```
            set2)
          set1))

(def ness
  (lambda (list)
          (cond ((and (null nessflag)
                      (mene (cur (caadar (last tree))) list))
                 (sete nessflag t))
                (t
                 (sete nessflag nil)))))

(makefore or (goal1 goal2)
  (subgoal '(goal1 goal2))
  (sub_not_on_move '(nil))
  (countergoal(list (goalinst 'and (list (car (geti goal1 'countergoal))
                                         (car (geti goal2 'countergoal)))))))

(makefore and (goal1 goal2)
  (subgoal '(goal1 goal2))
  (sub_not_on_move '(goal1 goal2))
  (countergoal (list (goalinst 'or (list (car (geti goal1 'countergoal))
                                         (car (geti goal2 'countergoal)))))))
(makefore andnot (goal1 goal2)
  (subgoal (allbut  'goal2 (eval (geti 'goal1 'subgoal))))
  (sub_not_on_move (eval (geti 'goal1 'sub_not_on_move)))
  (countergoal (eval (geti 'goal1 'countergoal)))
  (feasible (eval (geti 'goal1 'feasible)))
  (succeeded (eval (geti 'goal1 'succeeded)))
  (failed (eval (geti 'goal1 'failed))))

(makefore avoid (goal)
  (succeeded (eval (geti 'goal 'succeeded)))
  (failed (eval (geti 'goal 'failed))))
```

Nov 29 13:53 1984 oth_program.l Page 1

```
(def othello
  (lexpr (parm)
         (array board t 8 8)
         (setq xxxx nil)
         (init)
         (cond ((or (eq parm 0)
                    (null (arg 1)))
                (readstart '/u/14721ja/workingcpx/init))
               (t
                (readstart (arg 1))))
         (cond ((or (lessp parm 2)
                    (null (arg 2)))
                (load '/u/14721ja/workingcpx/othellogoals))
               (t
                (load (arg 2))))
         (do ((mc 1 (1+ mc))
              (mv))
             ((equal mc 60))
             (display)
             (move (readmove) 'black)
             (display)
             (princ "one moment")(terpri)
             (setq tree (search board))
             (playboard tree))))

(def readmove
  (lambda ()
          (prog (temp)
                loop
                (princ "black -- enter your move ")
                (setq temp (read))
                (cond ((eq temp 'r)
                       (newout tree)
                       (go loop))
                      ((eq temp 'b)
                       (display)
                       (go loop))
                      ((eq temp 't)
                       (pp tree)
                       (go loop))
                      ((eq temp 's)
                       (princ "name of file : ")
                       (savename (ratom))
                       (go loop))
                      ((and temp
                            (or (atom temp)
                                (not (eq (length temp) 2))
                                (not (validmove temp 'black))
                                (null temp)))
                       (princ "invalid move -- try again")
                       (terpri)
                       (go loop)))
                (return temp))))

(def move
  (lambda (space color)
```

```
        (cond (space
               (apply 'board (cons color space))
               'mapc '(lambda (k)
                              (and (flipable space (sublist k space) color)
                                   (flip space (sublist k space) color)))
                     (adjacent space))))))

(def flip
  (lambda (space dir color)
          (apply 'board (cons color (addlist space dir)))
          (cond ((equal (apply 'board (addlist (addlist space dir) dir))
                        (op color))
                 (flip (addlist space dir) dir color)))))

(def validmove
  (lambda (space color)
          (some '(lambda (k)
                         (flipable space (sublist k space) color))
                (adjacent space))))

(def flipable
  (lambda (space dir color)
          (cond ((and (validp (addlist space dir))
                      (equal (apply 'board (addlist space dir))
                             (op color)))
                 (flipable2 space dir color)))))

(def flipable2
  (lambda (space dir color)
          (and (validp (addlist space dir))
               (cond ((equal (apply 'board (addlist space dir))
                             (op color))
                      (flipable2 (addlist space dir) dir color))
                     ((equal (apply 'board (addlist space dir))
                             color)
                      (addlist space dir))))))
(def display
  (lambda ()
          (do ((i 0 (1+ i)))
              ((greaterp i 7))
              (do ((j 0 (1+ j)))
                  ((greaterp j 7))
                  (display1 i j)
                  (princ "   "))
              (princ i)
              (terpri)
              (terpri))
          (do ((j 0 (1+ j)))
              ((greaterp j 7))
              (princ j)
              (princ "       "))
          (terpri)))


(def display1
  (lambda (i j)
```

```
        (cond ((null (board i j))
               (princ '.....'))
              (t
               (princ (board i j))))))

(def init
  (lambda ()
        (sets $side 'countersoal)
        (sets pselist nil)
        (sets teampoals nil)
        (sets pempoals nil)))

(def readstart
  (lambda (filename)
        (sets pt(infile filename))
        (do ((i 0 (1+ i)))
            ((greaterp i 7))
            (do ((j 0 (1+ j)))
                ((greaterp j 7))
                (board (read pt) i j)))
        (close pt)))

(def savename
  (lambda (filename)
        (sets pt(outfile filename))
        (do ((i 0 (1+ i)))
            ((greaterp i 7))
            (do ((j 0 (1+ j)))
                ((greaterp j 7))
                (print (board i j) pt)
                (terpri pt)))
        (close pt)))

(def validp
  (lambda (subs)
        (and (lessp (car subs) 8)
             (greaterp (car subs) -1)
             (lessp (cadr subs) 8)
             (greaterp (cadr subs) -1))))

(def op
  (lambda (color)
        (cond ((equal color 'white)
               'black)
              (t
               'white))))

(def addlist
  (lambda (l1 l2)
        (nconc '(+ l1 l2))))

(def sublist
  (lambda (l1 l2)
        (nconc '- l1 l2))))

(def adjacent
```

```
  (lambda (squ)
          (subset 'valid? (mapcar '(lambda (k)
                                     (addlist k squ))
                                  '((1 0) (1 1) (0 1) (-1 1) (-1 0) (-1 -1)
                                    (0 -1) (1 -1))))))

(def two_away
  (lambda (squ)
          (subset 'valid? (mapcar '(lambda (k)
                                     (addlist k squ))
                                  '((2 0) (2 2) (0 2) (-2 2) (-2 0) (-2 -2)
                                    (0 -2) (2 -2))))))
(def cpy
  (nlambda (params)
           (*array (car params) t 8 8)
           (do ((i 0 (1+ i)))
               ((greaterp i 7))
               (do ((j 0 (1+ j)))
                   ((greaterp j 7))
                   (apply (car params)
                          (list (apply (cadr params) (list i j)) i j))))))

(def continue_play
  (lambda ()
          (setq xxxx (not xxxx))))

(def act
  (lambda (space color)
          (apply 'environment (cons color space))
          (mapc '(lambda (k)
                         (and (aflipable space (sublist k space) color)
                              (aflip space (sublist k space) color)))
                (adjacent space))))

(def aflip
  (lambda (space dir color)
          (apply 'environment (cons color (addlist space dir)))
          (cond ((equal (apply 'environment (addlist (addlist space dir) dir))
                        (op color))
                 (aflip (addlist space dir) dir color)))))

(def aflipable
  (lambda (space dir color)
          (cond ((and (valid? (addlist space dir))
                      (equal (apply 'environment (addlist space dir))
                             (op color)))
                 (aflipable2 space dir color)))))

(def aflipable2
  (lambda (space dir color)
          (and (valid? (addlist space dir))
               (cond ((equal (apply 'environment (addlist space dir))
                             (op color))
                      (aflipable2 (addlist space dir) dir color))
                     ((equal (apply 'environment (addlist space dir))
                             color)
```

```
                    (addlist space dir))))))

(def adisplay
  (lambda ()
          (do ((i 0 (1+ i)))
              ((greaterp i 7))
              (do ((j 0 (1+ j)))
                  ((greaterp j 7))
                  (adisplay1 i j)
                  (princ "   "))
              (princ i)
              (terpri)
              (terpri))
          (do ((j 0 (1+ j)))
              ((greaterp j 7))
              (princ j)
              (princ "       "))
          (terpri)))


(def adisplay1
  (lambda (i j)
          (cond ((null (environment i j))
                 (princ "....."))
                (t
                 (princ (environment i j))))))

(def avalidmove
  (lambda (space color)
          (some '(lambda (k)
                         (aflipable space (sublist k space) color))
                (adjacent space))))
```

Nov 29 13:49 1984 goal_util.l Page 1

```
'( This file contains all the variable definitions
   and othello functions that are explicitly called
   in the othello goal objects )


(setq midsqs '( (2 2) (2 3) (2 4) (2 5)
                (3 2) (3 3) (3 4) (3 5)
                (4 2) (4 3) (4 4) (4 5)
                (5 2) (5 3) (5 4) (5 5) ))

(setq sidesqs '( (0 2) (0 3) (0 4) (0 5)
                 (2 0) (3 0) (4 0) (5 0)
                 (7 2) (7 3) (7 4) (7 5)
                 (2 7) (3 7) (4 7) (5 7) ))

(setq nearsidesqs '( (1 2) (1 3) (1 4) (1 5)
                     (2 1) (3 1) (4 1) (5 1)
                     (6 2) (6 3) (6 4) (6 5)
                     (2 6) (3 6) (4 6) (5 6) ))

(setq cnrsqs '( (0 0) (0 7) (7 0) (7 7) ))

(setq sidenearcnr '( (0 1) (1 0) (1 7) (0 6)
                     (7 1) (6 0) (6 7) (7 6) ))

(setq notsidenearcnr '( (1 1) (1 6) (6 1) (6 6) ))

(setq nearcnr (append sidenearcnr notsidenearcnr))

(setq alldirs '((1 1) (1 0) (0 1) (1 -1) (-1 1) (-1 -1) (-1 0) (0 -1)))

'( returns list of spaces that are two_away from the square cnr
   and of the same color as side)
(def no_two_away
  (lambda (cnr side)
          (prog (lst)
                (mapcar '(lambda (k)
                                 (and (equal (environment (car k) (cadr k))
                                             side)
                                      (cons k lst)))
                        (two_away cnr))
                (return lst))))
'( indicates is there is a legal play two_away from position cnr for the
   color defined in side)
(def can_play_1_2_away
  (lambda (cnr side)
          (prog (flag)
                (mapcar '(lambda (k)
                                 (and (null (environment (car k) (cadr k)))
                                      (availdmove k side) (setq flag t)))
                        (two_away cnr))
                (return flag))))
'( finds a play in the direction of dir that will turn over
```

```
   the peice in the square pos)
(def turn_play
  (lambda (pos dir)
    (prog (side flg k1 k2 new)
          (setq side (environment (car pos) (cadr pos)))
          (setq new pos)
     loop (setq new (list (add (car new) (car dir))
                          (add (cadr new) (cadr dir))))
          (cond ((or (> (abs (car new)) 7)
                     (> (abs (cadr new)) 7)
                     (< (car new) 0)
                     (< (cadr new) 0)) (return nil))
                ((equal (environment (car new) (cadr new)) side)
                 (go loop))
                ((null (environment (car new) (cadr new)))
                 (setq k1 new))
                (t (setq k2 new)))
          (and (null flg) (setq flg t) (setq new pos)
               (setq dir (list (times -1 (car dir)) (times -1 (cadr dir))))
               (go loop))
          (cond ((or (null k1) (null k2)) (return nil))
                (t (return k1))))))

'( finds all legal moves that will turn_over the piece in the pos square)
(def turn_moves
  (lambda (pos)
    (prog (lst)
          (mapcar '(lambda (k)
                           (and (setq k (turn_play pos k))
                                (setq lst (cons k lst))))
                  '((1 1) (1 -1) (1 0) (0 1)))
          (return lst))))

'( finds a play in direction dir that will allow the piece in position pos
   to be turned over on the next play )
(def set_up_turn_play
  (lambda (pos dir)
    (prog (side flg k1 k2 k3 new)
          (setq side (environment (car pos) (cadr pos)))
          (setq new pos)
     loop (setq new (list (add (car new) (car dir))
                          (add (cadr new) (cadr dir))))
          (cond ((or (> (abs (car new)) 7)
                     (> (abs (cadr new)) 7)
                     (< (car new) 0)
                     (< (cadr new) 0)) (return nil))
                ((equal (environment (car new) (cadr new)) side)
                 (and (setq k3 (cons new k3)) (go loop)))
                ((null (environment (car new) (cadr new)))
                 (setq k1 (cons new k1))))
          (and (null flg) (setq flg t) (setq new pos)
               (setq dir (list (times -1 (car dir)) (times -1 (cadr dir))))
               (go loop))
          (return (append k3 k1)))))

'( )
```

```
(def set_up_turn_moves
  (lambda (pos)
    (prog (lst)
          (mapcar '(lambda (k)
                           (and (setq lst (append lst
                                                  (set_up_turn_play pos k)))))
                  '((1 1) (1 -1) (1 0) (0 1)))
          (return lst))))

'( set_up_moves  finds the set of all squares that if side could control then
                 the goal square could be occupied on the next turn)
(def set_up_moves
  (lambda (cnr side)
    (prog (lst1 lst2)
      (or (setq lst1 (turn_targets (one_away cnr) (op side))) (return nil))
      loop
      (setq lst2 (append
                  (set_up_turn_play (car lst1)
                                    (sub_pos (car lst1) cnr))
                  lst2))
      (and (setq lst1 (cdr lst1)) (go loop))
      (return lst2))))

'( legalsq?  tests if given pos is a legal one)
(def legalsq?
  (lambda (k)
    (and (< (car k) 8)
         (> (car k) -1)
         (< (cadr k) 8)
         (> (cadr k) -1))))

'( one_away returns all legal squares one position away from
   specified position)
(def one_away
  (lambda (k1)
    (prog (lst)
      (setq lst
            (mapcar
              '(lambda (k2) (list (add (car k1) (car k2))
                                  (add (cadr k1) (cadr k2))))
              '((1 1) (1 0) (0 1) (1 -1) (-1 1) (-1 -1) (-1 0) (0 -1))))
      (setq lst (subset 'legalsq? lst))
      (return lst))))

'( turn_targets  given a list of board positions returns those occupied
   by the specified side)
(def turn_targets
  (lambda (k1 side)
    (prog (lst)
      (mapcar
        '(lambda (k2)
           (and (equal (env k2) side) (setq lst (cons k2 lst))))
        k1)
      (return lst))))

'( env  is a single assignment version of environment)
```

```
(def env
  (lambda (k)
    (environment (car k) (cadr k))))

'( this is not a good utility)
(def env_test
  (lambda (k)
    (and (null (env k)) (validmove k 'black))))

'( add_pos  input = two equal size list of numbers
           output = list of sums of numbers in the input lists)
(def add_pos
  (lambda (k1 k2)
     (prog (lst)
       loop
       (setq lst (cons (add (car k1) (car k2)) lst))
       (setq k1 (cdr k1))
       (setq k2 (cdr k2))
       (and k1 (go loop))
       (return (reverse lst)))))

'(sub_pos   input = two equal size list of numbers
           output = list of subtractions of second numb set from first)
(def sub_pos
  (lambda (k1 k2)
     (prog (lst)
       loop
       (setq lst (cons (diff (car k1) (car k2)) lst))
       (setq k1 (cdr k1))
       (setq k2 (cdr k2))
       (and k1 (go loop))
       (return (reverse lst)))))

'( one_away returns all legal squares one position away from
   specified position)
(def one_away
  (lambda (k1)
    (prog (lst)
      (setq lst
            (mapcar
              '(lambda (k2) (list (add (car k1) (car k2))
                                  (add (cadr k1) (cadr k2))))
              alldirs))
      (setq lst (subset 'legalsq? lst))
      (return lst))))

'( piece_ptrn    input = position and a direction
                 output = pattern of black white and blank squares in direction)
(def piece_ptrn
  (lambda (pos dir)
    (prog (new lst1 lst2 tmp)
          (setq new pos)
          loop
          (setq new (add_pos new dir))
          (cond
            ((legalsq? new) (setq lst1 (cons (list (env new) new) lst1))
```

```
                                        (go loop)))
          loop2
          (cond
            ((null lst1) (return lst2))
            ((null lst2) (setq lst2 (cons (car lst1) lst2)))
            ((equal (caar lst1) (caar lst2))
               (setq tmp (cons (cadar lst1) (cdar lst2)))
               (setq tmp (cons (caar lst2) tmp))
               (setq lst2 (cons tmp (cdr lst2))))
            (t (setq lst2 (cons (car lst1) lst2))))
          (setq lst1 (cdr lst1))
          (go loop2))))

'( turn_to_save   input = position direction and side to save square for
                  output = location of op side piece to be turned over or nil)
(def turn_to_save
  (lambda (pos dir side)
    (prog (lst)
      (setq lst (piece_ptrn pos dir))
      (cond ((and (equal (caar lst) side) (equal (caadr lst) (op side)))
               (return (cadadr lst)))
            (t (return nil))))))

'( turn_all_to_save   input =  position to prevent other side from playing in
                     output =  list of positions of op side pieces to turn over)
(def turn_all_to_save
  (lambda (pos side)
    (delete nil
            (mapcar
              '(lambda (k) (turn_to_save pos k side))
              alldirs))))

'( delete_list  deletes elements of the first list
                from the second list)
(def delete_list
  (lambda (lst1 lst2)
    (prog ()
      (mapcar '(lambda (k)
                       (setq lst2 (delete k lst2)))
              lst1)
      (return lst2))))
```

Nov 29 13:50 1984 oth_goals.l Page 1

```
(setq pselist (mapcar
                '(lambda (k)
                        (goalinst 'play k))
                (subset 'cnr_test '((0 0) (0 7) (7 0) (7 7)))))

(def topgoalpair
  (lambda ()
         (list 'improve_position (goalinst 'prevent '(improve_position)))))

'( this is a generic goal for the conjunction of two specific goals )
(makeform and (goal1 goal2)
  (subgoal '(goal1 goal2))
  (sub_not_on_move '(goal1 goal2))
  (countergoal (list (goalinst 'or (list (car (geti goal1 'countergoal))
                                         (car (geti goal2 'countergoal)))))))

'( this is a generic goal for the disjunction of two specific goals )
(makeform or (goal1 goal2)
  (subgoal '(goal1 goal2))
  (sub_not_on_move '(nil))
  (countergoal(list (goalinst 'and (list (car (geti goal1 'countergoal))
                                         (car (geti goal2 'countergoal)))))))

'( top level goal for othello play)
(makegoal improve_position
  (countergoal (list (goalinst 'prevent '(improve_position))))
  (subgoal '(atck_a_cnr imp_a_cnr play_side play_middle play_any))
  (sub_not_on_move '(atck_a_cnr imp_a_cnr play_side play_middle play_any)))

'( this is a generic goal that instantiates only countergoals )
(makeform prevent (gname)
  (countergoal (list gname))
  (subgoal nil))

'( othello goal to play in a corner )
(makegoal atck_a_cnr
  (countergoal (list 'dfnd_a_cnr))
  (subgoal (mapcar '(lambda (k)
                          (goalinst 'atck_cnr k))
                   '( (0 0) (0 7) (7 0) (7 7))))
  (sub_not_on_move (mapcar '(lambda (k)
                                  (goalinst 'atck_cnr k))
                           '( (0 0) (0 7) (7 0) (7 7)))))

'( counter goal to stop atck_a_cnr from playing in a corner )
(makegoal dfnd_a_cnr
  (countergoal (list 'atck_a_cnr))
  (subgoal nil)
  (sub_not_on_move (mapcar '(lambda (k)
                                  (goalinst 'dfnd_cnr k))
                           '((0 0) (0 7) (7 0) (7 7)))))

'( positional goal to try to better position around a corner )
(makegoal imp_a_cnr
  (countergoal (list 'stop_a_cnr))
```

```
  (subgoal (mapcar '(lambda (k)
                      (goalinst 'imp_cnr k))
                   '( (0 0) (0 7) (7 0) (7 7) )))
  (sub_not_on_move (mapcar '(lambda (k)
                              (goalinst 'imp_cnr k))
                           '( (0 0) (0 7) (7 0) (7 7) ))))

'( counter goal to stop imp_a_cnr from getting a better corner position )
(makegoal stop_a_cnr
  (countergoal (list 'imp_a_cnr))
  (subgoal nil)
  (sub_not_on_move (mapcar '(lambda (k)
                              (goalinst 'stop_cnr k))
                           '( (0 0) (0 7) (7 0) (7 7) ))))

'( instantiates atck_a_cnr against a specific corner )
(makeform atck_cnr (cor1 cor2)
  (countergoal (list (goalinst 'dfnd_cnr '(cor1 cor2))))
  (subgoal (cons (goalinst 'play '(cor1 cor2))
                 (mapcar '(lambda (k)
                            (goalinst 'play_safe k))
                         (set_up_moves (list cor1 cor2) curside))))
  (sub_not_on_move nil)
  (failed (equal (environment cor1 cor2) (op curside))))

'( counter goal of atck_a_cnr )
(makeform dfnd_cnr (cor1 cor2)
  (countergoal (list (goalinst 'atck_cnr '(cor1 cor2))))
  (subgoal (list (goalinst 'play '(cor1 cor2))
                 (goalinst 'stop_play '(cor1 cor2))))
  (sub_not_on_move nil)
  (failed (equal (environment cor1 cor2) (op curside))))

'( instatniates imp_a_cnr against a specific corner)
(makeform imp_cnr (cor1 cor2)
  (countergoal (list (goalinst 'stop_cnr '(cor1 cor2))))
  (subgoal (list (goalinst 'ctrl_1_2_away '(cor1 cor2))))
  (sub_not_on_move (list (goalinst 'ctrl_1_2_away '(cor1 cor2))))
  (feasible (and (null (environment cor1 cor2))
                 t)))

'( counter goal of imp_cnr )
(makeform stop_cnr (cor1 cor2)
  (countergoal (list (goalinst 'ctrl_cnr '(cor1 cor2))))
  (subgoal (list (goalinst 'play '(cor1 cor2))
                 (goalinst 'stop_1_2_away '(cor1 cor2))))
  (sub_not_on_move (list (goalinst 'play '(cor1 cor2))
                         (goalinst 'stop_1_2_away '(cor1 cor2)))))

'( goal to play on a space that is one space removed from a corner )
(makeform ctrl_1_2_away (cor1 cor2)
  (countergoal (list (goalinst 'stop_1_2_away '(cor1 cor2))))
  (subgoal
           (mapcar '(lambda (k)
                      (cond ((member k sidesos)
                             (goalinst 'play_safe k))
```

```
                                    (t
                                     (goalinst 'not_nrcnr_play_safe k))))
                        (two_away '(cor1 cor2))))
  (sub_not_on_move nil))

'( counter goal to ctrl_1_2_away )
(makeform stop_1_2_away (cor1 cor2)
  (countergoal (list (goalinst 'ctrl_1_2_away '(cor1 cor2))))
  (subgoal nil)
  (sub_not_on_move (mapcar '(lambda (k)
                                    (cond ((member k sideses)
                                           (goalinst 'play_safe k))
                                          (t
                                           (goalinst 'not_nrcnr_play_safe k))))
                           (two_away '(cor1 cor2)))))

'( goal to play a piece on one of the side squares )
(makegoal play_side
  (countergoal (list 'stop_play_side))
  (subgoal (mapcar '(lambda (k)
                            (goalinst 'play_safe k))
                   sidesqs))
  (sub_not_on_move nil))

'( counter goal of play_side to prevent playing on the side )
(makegoal stop_play_side
  (countergoal (list 'play_side))
  (subgoal nil)
  (sub_not_on_move (mapcar '(lambda (k)
                            (goalinst 'play_safe k))
                           sideses)))

'( goal to play in the middle which is generally a safe, nonaggressive play )
(makegoal play_middle
  (countergoal '(nil))
  (subgoal (mapcar '(lambda (k)
                            (goalinst 'play k))
                   midses)))

'( play_safe is a goal to play on a square, and not be turned over )
(makeform play_safe (sp1 sp2)
  (countergoal (list (goalinst 'play_safe '(sp1 sp2))))
  (subgoal (list (goalinst 'play '(sp1 sp2))
                 (goalinst 'turn_over '(sp1 sp2))))
  (sub_not_on_move (list (goalinst 'turn_over '(sp1 sp2))))
  (feasible (or (and (equal (environment sp1 sp2) (op curside))
                     (turn_moves '(sp1 sp2)))
                (and (null (environment sp1 sp2))
                     (validmove '(sp1 sp2) curside))))
  (failed (and (equal (environment sp1 sp2) (op curside))
               (null (turn_moves '(sp1 sp2))))))

'( like play_safe but excludes plays next to a corner position )
(makeform not_nrcnr_play_safe (sp1 sp2)
  (countergoal (list (goalinst 'not_nrcnr_play_safe '(sp1 sp2))))
```

```
  (subgoal (list (goalinst 'play '(sp1 sp2))
                 (goalinst 'not_nrcnr_turn_over '(sp1 sp2))))
  (sub_not_on_move (list (goalinst 'not_nrcnr_turn_over '(sp1 sp2))))
  (feasible (or (and (equal (environment sp1 sp2) (op curside))
                     (delete_list nearcnr (turn_moves '(sp1 sp2))))
                (and (null (environment sp1 sp2))
                     (validmove '(sp1 sp2) curside))))
  (failed (and (equal (environment sp1 sp2) (op curside))
               (null (delete_list nearcnr (turn_moves '(sp1 sp2)))))))

'( goal to prevent opponent from play on a square on the next move )
(makeform stop_play (cor1 cor2)
  (countergoal nil)
  (subgoal (mapcar '(lambda (k)
                      (goalinst 'turn_over k))
                   (turn_all_to_save (list cor1 cor2) curside)))
  (sub_not_on_move nil))

'( like turn_over, but does not allow play next to corner )
(makeform not_nrcnr_turn_over (sp1 sp2)
  (countergoal (list (goalinst 'not_nrcnr_turn_over '(sp1 sp2))))
  (subgoal (mapcar '(lambda (k)
                      (goalinst 'play k))
                   (delete_list nearcnr (turn_moves '(sp1 sp2)))))
  (sub_not_on_move nil)
  (feasible (delete_list nearcnr (turn_moves '(sp1 sp2)))))

'( goal to turn_over a piece on a square )
(makeform turn_over (sp1 sp2)
  (countergoal '(nil))
  (subgoal (mapcar '(lambda (k)
                      (goalinst 'play k))
                   (turn_moves '(sp1 sp2))))
  (feasible (turn_moves '(sp1 sp2))))

'( if nothing else works, then play anywhere )
(makegoal play_any
  (subgoal (mapcar '(lambda (k)
                      (goalinst 'play k))
                   (append midsqs
                           (delete_list nearcnr nearsidesqs)
                           (delete_list nearcnr sidesqs)
                           nearcnr))))

'( goal to simply play on a square )
(makeform play (sp1 sp2)
  (countergoal '(nil))
  (feasible (and (null (environment sp1 sp2))
                 (validmove '(sp1 sp2) curside)))
  (action (act '(sp1 sp2) curside))
  (updateaction (move '(sp1 sp2) curside)))
```